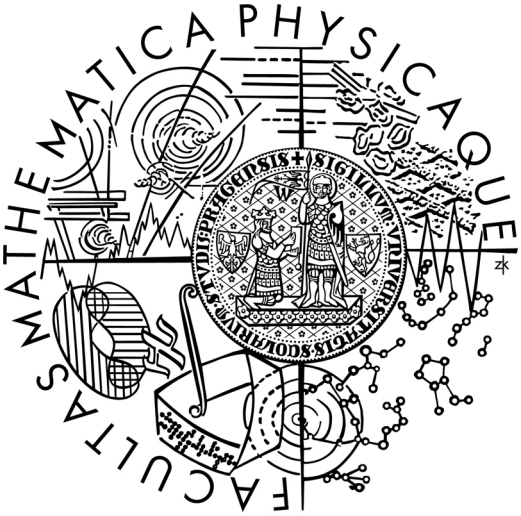
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Pavol Gajarský

Generování kódu z diagramů aktivity jazyka UML

Katedra softwarového inženýrství
Vedoucí diplomové práce: *doc. Ing. Karel Richta, CSc.*
Studijní program: *Informatika, Softwarové systémy, Softwarové inženýrství*
Praha 2011

Poďakovanie

V prvom rade by som chcel poďakovať vedúcemu svojej diplomovej práce doc. Ing. Karlovi Richtovi, CSc. za podnetné rady a pripomienky a predovšetkým za venovaný čas.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 4.8.2011 Pavol Gajarský

Název práce: *Generování kódu z diagramů aktivity jazyka UML*
Autor: *Pavol Gajarský*
Katedra (ústav): *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *doc. Ing. Karel Richta CSc.*
e-mail vedoucího: *richta@ksi.mff.cuni.cz*
Abstrakt: *Generování kódu z jazyka UML je dosud značně omezené. Diagramy, z nichž je nejčastěji generován kód popisují statickou strukturu. Jde především o diagramy tříd. Jedním z nejužitečnějších a nejméně využitých druhů jsou právě diagramy popisující chování. Příčinou toho je především nedostatečná podpora ze strany specifikace UML. Až nástup verze 2.0 přinesl viditelné změny v této oblasti. Tato diplomová práce bude zaměřena pouze na diagramy aktivity. Jejím cílem bude navrhnout způsob generování kódu z tohoto druhu diagramů. Návrh bude ověřen na prototypu nástroje, který dokáže generovat kostru kódu v dohodnutém jazyce (např. Java, C + +, PHP,…). Následně bude srovnán s alternativními přístupy a zhodnocen z hlediska jeho výhod a nevýhod. Jako vstup se předpokládá diagram aktivity vytvořen a vyexportovaný externím CASE nástrojem ve formátu XMI. Požadavkem na generátor kódu bude zvolení architektury umožňující přidávání nových modulů. Tedy koncovému uživateli bude zprostředkované rozhraní, jehož prostřednictvím si bude moci doprogramovat další výstupní jazyk.*
Klíčová slova: *Generování, kód, UML, diagramy, aktivity*

Title: *Code Generation from UML Activity Diagrams*
Author: *Pavol Gajarský*
Department: *Department of Software Engineering*
Supervisor: *doc. Ing. Karel Richta CSc.*
Supervisor's e-mail address: *richta@ksi.mff.cuni.cz*
Abstract: *Code generation from UML is still very limited. Diagrams, which are usually used for code generation describes the static structure. These are mostly the class diagrams. One of the most useful and least used types of diagrams is behavioral diagrams. The reason for this is mainly a lack of support from the UML specification. The onset of version 2.0 has brought visible changes in this area. This thesis will focus only on activity diagrams and possibilities of code generation from them. One of them will be verified on a prototype tool that can generate skeleton of code in chosen language (e.g. Java, C + +, PHP ,...). Then it will be compared to alternative approaches and evaluated in terms of its advantages and disadvantages. As input is expected activity diagram created and exported by external CASE tool in XMI format. Architecture of tool will provide possibility of defining new modules. So the end user will be able to program another output language by the medium of given interface.*
Keywords: *Generation, code, UML, diagrams, activity*

Obsah

1. Úvod

Generovanie kódu z jazyka UML je doposiaľ značne obmedzené. Diagramy, z ktorých je najčastejšie generovaný kód popisujú statickú štruktúru. Ide predovšetkým o diagramy tried. Jedným z najužitočnejších a najmenej využitých druhov sú práve diagramy popisujúce správanie, keďže prostredníctvom nich je možné modelovanie správania elementov v čase a priestore. Príčinou toho je predovšetkým nedostatočná podpora zo strany špecifikácie UML. Až nástup verzie 2.0 priniesol viditeľné zmeny v tejto oblasti. Medzi najhlavnejšie z nich patrí rozšírenie metamodelu UML o akcie, prostredníctvom ktorých je možné reprezentovať jednotlivé elementy diagramov správania. Tým je definovaná reprezentácia slúžiaca ako medzikód. Táto diplomová práca bude zameraná jedine na diagramy aktivity. V nasledujúcom zhrnieme jednotlivé výhody a nevýhody generovania kódu

Výhody

- Vysoká efektivita pri implementácii opakujúcich sa častí.
- Umožňuje nenáročnú zmenu implementácie generovaných častí aplikácie. Zmenu je nutné uskutočniť jedine v rámci generátora, následné zmeny v implementácii sa prejavia po pregenerovaní kódu všetkých dotknutých elementov. Nie sú nutné žiadne manuálne úpravy mimo generátora.
- Zníženie chybovosti v dôsledku toho, že výsledný kód je pre tú istú situáciu vygenerovaný rovnako. Tým dochádza k eliminácii chýb, ktoré môžu vzniknúť nedopatrením pri implementácii.
- Sústredenie pozornosti na samotný vývoj a riešenie logickej časti aplikácie namiesto riešenia technických problémov a ručného písania kódu.
- Čistota a konzistencia kódu spôsobená generovaním v rovnakom formáte, použitím rovnakého štýlu komentárov, použitím rovnakej notácie pri pomenovaní premenných ...
- Zrýchlenie vývojového procesu vygenerovaním kostry kódu aplikácie.

Nevýhody

- Generátor kódu musí byť napísaný ako prvý, čo typicky vyžaduje relatívne veľký objem práce.
- Nie je ho možné použiť v každej situácii, hodí sa skôr na opakujúce sa práce z minulosti, na ktoré je jasne definovaný postup riešenia.

1.1. Vytýčené ciele

Cieľom tejto sekcie je sformulovanie hlavných cieľov diplomovej práce, ktorými sú

- Popis samotnej problematiky generovania kódu z diagramov aktivity. Zhodnotenie toho, či je tento typ diagramov vhodný a užitočný pre generovanie kódu.
- Analýza možných prístupov pri generovaní kódu.
- Zvolenie jednej z popísaných metód, ktorá bude následne použitá pri implementácii prototypu umožňujúceho generovanie kostry kódu z diagramu aktivity vyexportovaného do formátu XMI. Tým sa následne overí správnosť daného prístupu.
- Porovnanie implementovaného nástroja s existujúcimi riešeniami. Zhodnotenie jeho výhod a nevýhod.

1.2. Popis jednotlivých kapitol

Cieľom tejto sekcie je popis jednotlivých kapitol. Tie sú radené tak, aby oboznámili čitateľa s diagramom aktivity jazyka UML, samotnou problematikou generovania kódu z tohto typu diagramu a nakoniec popísali jednotlivé prístupy použiteľné za daným účelom.

- *Úvod* – obsahuje popis vytýčených cieľov.
- *UML* – obsahuje popis jazyka UML a jeho jednotlivých diagramov.
- *Diagramy aktivity* - obsahuje popis diagramov aktivity a ich jednotlivých elementov, ktoré budú predmetom generovania kódu.
- *XMI* – obsahuje popis formátu XMI slúžiaceho ako vstupná reprezentácia diagramu aktivity pri generovaní kódu. V úvode kapitoly je zameraná pozornosť na formát XML, prostredníctvom ktorého je zapísaný samotný obsah XMI formátu. Ten odpovedá metamodelu, ktorý je možné vyjadriť prostredníctvom MOF. Z tohto dôvodu je záver kapitoly venovaný práve štandardu MOF.
- *Generovanie kódu* – obsahuje popis obecného prístupu pri generovaní kódu z diagramov aktivity. Ten je výsledkom snáh UML 2.0 o zavedenie reprezentácie diagramov správania prostredníctvom akcií (súčasť metamodelu UML), ako základných stavebných prvkov.
- *Vlastné generovanie kódu* – obsahuje popis zvoleného prístupu pri generovaní kódu, ktorého implementáciu reprezentuje samotný prototyp nástroja.
- *Alternatívne prístupy pri generovaní kódu* – obsahuje popis alternatívnych prístupov pri generovaní kódu pomimo použitej metódy pri implementácii prototypu.
- *Prototyp nástroja* – obsahuje popis užívateľského rozhrania prototypu nástroja dodaného k diplomovej práci.
- *Existujúce nástroje* – obsahuje porovnanie prototypu s existujúcimi nástrojmi rovnakého alebo podobného zamerania.

- *Záver* – obsahuje zhodnotenie vytýčených cieľov v úvode práce.
- *Zoznam použitej literatúry* – obsahuje odkazy na použitú literatúru a ostatné zdroje informácií.
- *Prílohy* – obsahuje jednotlivé prílohy k diplomovej práci.

2. UML

2.1. Úvod

UML (Unified Modeling Language) je vizuálny modelovací jazyk určený pre návrh softvérových systémov. Častejšie je však spájaný s modelovaním objektovo orientovaných systémov. Pri jeho návrhu bol kladený dôraz na to, aby v ňom boli uplatnené najlepšie praktiky používané v softvérovom inžinierstve. UML však nie je zviazaný so žiadnou konkrétnou metodológiou, jeho hlavným cieľom je poskytnúť vizuálnu syntax, prostredníctvom ktorej je možné modelovať. UML teda nezodpovedá na otázky, aké typy modelov majú byť pre daný systém vytvorené, prípadne v ktorej fáze vývoja. Jeho použitie je teda nezávislé od používanej metodológie vývoja. Pri návrhu bol taktiež kladený dôraz na jeho použitie pri modelovaní systémov rôzneho zamerania, pričom výsledok by bol platformovo a jazykovo nezávislý. Takto navrhnutý model je potom možné implementovať použitím rôznych technológií ako napr. Java alebo .NET. Základnou myšlienkou UML je modelovanie systémov ako kolekcie navzájom prepojených a spolupracujúcich objektov, čo vystihuje architektúru objektovo orientovaných systémov. Výsledné modely sú ľuďmi ľahko čitateľné a prinášajú formalizmus v komunikácii medzi širokou škálou ľudí vrátane cieľového zákazníka navrhovaného systému. Samotný proces modelovania je možné realizovať prostredníctvom nástrojov softvérového inžinierstva tzv. CASE nástrojov.

Jazyk UML má dva aspekty

- *Statická štruktúra* - popisuje aké typy elementov sú dôležité pre daný systém a akým spôsobom sú navzájom prepojené.
- *Popis správania* - popisuje životný cyklus týchto elementov a ich vzájomnú spoluprácu, aby zabezpečili požadovanú funkcionalitu systému.

Oba aspekty sa navzájom dopĺňajú a sú nevyhnutné pre kompletný popis systému.

2.2. Metamodel UML

Metamodel UML tvoria 3 časti

- Základné stavebné prvky.
- Pravidlá používané pri zostavovaní modelu.
- Mechanizmy.

2.2.1. Základné stavebné prvky

Medzi základné stavebné prvky UML patria

- Elementy (Things).
- Vzťahy (Relationships).
- Diagramy (Diagrams).

Elementy

Sú základné objektovo orientované bloky UML, ktoré reprezentujú abstrakcie v modely. Súvisiace elementy sú navzájom prepojené prostredníctvom väzieb a združované do diagramov.

UML rozlišuje nasledujúce typy elementov

- *Štrukturálne elementy (Structural elements)* – zahŕňajú prevažne statické časti modelu. Súhrnne sa označujú ako classifiers. Radia sa medzi ne napr. class, interface, collaboration, use case, active class, artifact a node.
- *Elementy správania (Behavioral elements)* - zahŕňajú dynamické časti modelu. Popisujú správanie prvkov modelu v čase a priestore. Radia sa medzi ne interaction, state machine a action.
- *Skupinové elementy (Grouping elements)* - slúžia na organizáciu modelu. Tá je realizovaná prostredníctvom dekompozície modelu do balíkov. Táto skupina obsahuje jediný element, ktorým je balík (package).
- *Anotačné elementy (Annotational elements)* - zahŕňajú vysvetľujúce časti modelu. Slúžia na priradenie popisu, vysvetlenia alebo poznámky k ľubovoľnému elementu modelu. Táto skupina obsahuje jediný element, ktorým je note.

Vzťahy

Tvoria základné relačné stavebné prvky jazyka UML. Slúžia na prepojenie jednotlivých elementov modelu.

UML rozlišuje 4 základné druhy vzťahov

- *Závislosť (Dependency)* – relácia, v ktorej zmena nezávislého elementu môže spôsobiť zmenu sémantiky elementu, ktorý je od neho závislý.
- *Asociácia (Association)* – všeobecná relácia medzi elementmi, ktoré sú navzájom prepojené. Môže obsahovať pomenovanie vzťahu a jeho kardinalitu.
- *Generalizácia (Generalization)* – relácia, v ktorej jeden element (potomok) je špecializáciou druhého (predok). Špecializovaný element preberá od svojho predka ako

štruktúru, tak i popis správania (vyjadreného napríklad prostredníctvom diagramu aktivity). V OOP je možné týmto vzťahom rozumieť dedičnosť.

- *Realizácia (Realization)* – relácia, v ktorej jeden element je realizáciou druhého, teda jeden z elementov špecifikuje kontrakt, ktorý druhý element musí naplniť. Tento vzťah typicky nastáva medzi triedou a rozhraním, ktoré má implementovať.

Diagramy

Diagram je grafické zobrazenie množiny navzájom súvisiacich elementov. Najčastejšie je vo forme orientovaného grafu, v ktorom vrcholy sú tvorené elementmi a hrany vzťahmi. Hrana medzi dvoma elementmi vedie v prípade, ak medzi nimi existuje nejaká forma vzťahu. V samotnom diagrame sa môže teoreticky vyskytovať ľubovoľná kombinácia elementov a vzťahov, avšak z praktického hľadiska nie sú všetky kombinácie zmysluplné. Z daného dôvodu rozlišuje UML 13 rôznych typov diagramov, ktoré je možné rozdeliť do 2 kategórií.

Diagramy popisujúce štruktúru

- Diagramy tried (Class diagrams).
- Diagramy objektov (Object diagrams).
- Diagramy štruktúry balíkov (Package diagrams).
- Diagramy štruktúr (Composite structure diagrams).
- Diagramy komponent (Component diagrams).
- Diagramy nasadenia (Deployment diagrams).

Diagramy popisujúce správanie

- Diagramy použitia (Use Case diagrams).
- Diagramy aktivity (Activity diagrams).
- Diagramy interakcie
    - Sekvenčné diagramy (Sequence diagrams).
    - Stavové diagramy (State machine diagrams).
    - Diagramy komunikácie (Communication diagrams).
    - Prehľadové diagramy interakcie (Interaction overview diagrams).
    - Časové diagramy (Timing diagrams).

2.2.2. Pravidlá

Jednotlivé časti UML nemôžu byť používané ľubovoľne, keďže takto vzniknutý model by nemusel dávať zmysel. Obmedzenia kladené pri vytváraní modelu stanovuje UML prostredníctvom pravidiel. Dodržiavanie týchto pravidiel pri modelovaní zaručuje vytvorenie validného modelu.

UML definuje sémantické pravidlá pre nasledujúce oblasti

- *Mená (Names)* - definujú, čo všetko je možné označiť za element, vzťah alebo diagram.
- *Rozsah (Scope)* – definujú kontext, v ktorom dané meno nadobúda špecifický význam.
- *Viditeľnosť (Visibility)* – definujú, za akých podmienok môžu byť jednotlivé súčasti UML modelu viditeľné a používané v iných častiach.
- *Obmedzenia (Integrity)* – definujú, ako môžu byť objekty navzájom správne a konzistentne prepojené.
- *Vykonávanie (Execution)* – definujú, čo presne znamená vykonanie alebo simulácia dynamického modelu.

2.2.3. Mechanizmy

Táto časť metamodelu UML obsahuje mechanizmy pomáhajúce zaistiť konzistenciu výsledného modelu. UML obsahuje mechanizmy pre nasledujúce oblasti

- *Špecifikácia (Specification)* - slúži na popis sémantiky jednotlivých elementov alebo častí modelu.
- *Ornamenty (Adornments)* - slúžia na rozšírenie elementu o detaily, ktoré nie je možné zachytiť prostredníctvom jeho atribútov. Môžu mať textovú alebo vizuálnu podobu.
- *Bežné delenie (Common divisions)* - slúži na oddelenie tried od ich inštancií alebo rozhrania od samotnej implementácie.
- Rozširovateľnosť (Extensibility mechanisms) - umožňuje rozšírenie jazyka o nové stereotypy, príznaky a obmedzenia.

3. Diagramy aktivity

3.1. Úvod

Patria medzi diagramy UML popisujúce správanie. Často sa považujú za obdobu stavových diagramov, v ktorých sú stavy nahradené aktivitami, pričom prechod medzi nimi nastáva až po ukončení aktuálne prebiehajúcej aktivity. Typicky sa vzťahujú k jednému prípadu použitia alebo metóde objektu, ale je možné ich priradenie k ľubovoľnému elementu UML za účelom modelovania jeho správania v čase a priestore. Nezriedka sú prostredníctvom nich modelované business procesy a workflows. Diagram aktivity je reprezentovaný grafom, ktorého vrcholy tvoria jednotlivé elementy a hrany sú realizované prostredníctvom dátového toku a toku riadenia.

3.2. Popis

Pri modelovaní diagramov aktivity je možné použiť nasledujúce elementy

- Aktivity (Activity).
- Štruktúrované aktivity (Structured activity).
- Počiatočný a koncový stav (Initial and Final action state).
- Akcie (Actions).
- Prechody (Flow transitions a Object flow).
- Rozhodnutia a zjednotenia (Decisions and Merges).
- Vetvenia a spojenia (Forks and Joins).
- Oblasti zodpovednosti (Swimlanes).
- Signály (Signals).
- Časové udalosti (Time events).
- Rozširujúci región (Expansion region).
- Piny (Pins).
- Výnimky (Exceptions).
- Vstupné a výstupné podmienky (Preconditions and Postconditions).

3.2.1. Aktivity

Aktivity (Activity) sú užívateľom definované elementy diagramov aktivity s najnižšou granularitou. Reprezentujú vykonávanú činnosť v danom momente. Ich vykonanie reprezentuje určitú transformáciu alebo proces v modelovanom systéme. UML nedefinuje obsah aktivít, teda môžu obsahovať ako obyčajný text, tak i príkazy konkrétneho programovacieho jazyka. V prípade, že do aktivity vstupuje viac prechodov, je vykonaná až potom, ako sú aktivované všetky jej vstupy. Aktivity reprezentujú užívateľom definované správanie, ktoré môže byť vyvolané príslušnou akciou s ktorou si môže vymieňať dáta. Aktivity a všetky užívateľom definované správania sú v UML realizované ako triedy, ktoré môžu byť parametrizované. V momente, keď je daná aktivita vykonávaná, je vytvorená inštancia príslušnej triedy, ktorá ju reprezentuje. Po skončení jej vykonávania je daná inštancia zničená.

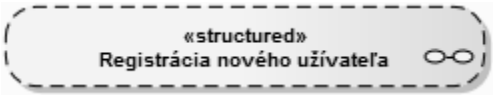
Vlastnosti aktivít

- *Sú atomické* - nie je možné ich ďalšie delenie.
- *Sú neprerušiteľné* – ak tok riadenia vstúpi do niektorej z aktivít, pokračuje sa v ďalšom vykonávaní až po jej ukončení.
- *Sú okamžité* - potrebujú na svoje vykonanie zanedbateľné množstvo času.

V UML sú zobrazené prostredníctvom obdĺžnikov so zaoblenými rohmi.

3.2.2. Štruktúrované aktivity

Štruktúrované aktivity (Structured activity) reprezentujú vykonávané činnosti, ktoré nie sú atomické, teda ich vykonávanie zaberá určitý čas. Môžu byť delené do ďalších aktivít alebo subaktivít. Sú reprezentované samostatným diagramom aktivity. Typicky sa používajú pre zjednodušenie modelu tým, že sa detaily, ktoré sú v danom kontexte nepodstatné, skryjú do príslušných štruktúrovaných aktivít. Sú jedinými elementmi, ktoré môžu obsahovať vo svojom vnútri prepojenie iných elementov vrátane ďalších štruktúrovaných aktivít. Podobne ako obyčajná aktivita i štruktúrovaná môže byť súčasťou toku riadenia a dátového toku, a teda i obsahovať príslušné vstupy. Po skončení vykonávania štruktúrovanej aktivity je tok riadenia predaný aktivitám, ktoré sú s ňou spojené. Štruktúrované aktivity je vhodné používať za účelom dosiahnutia modularizácie modelu, ktorá môže byť následne v cieľovom jazyku zrealizované prostredníctvom implementácie odpovedajúcich metód.



3.2.3. Počiatočný a koncový stav

Diagramy aktivity obsahujú dva špeciálne stavy určené na označenie začiatku a konca vykonávaných činností. Každý diagram aktivity obsahuje aspoň jeden počiatočný (Initial state) a koncový (Final state) stav. Tok riadenia začína práve v počiatočnom stave. Ak diagram obsahuje viac počiatočných stavov, tak tok riadenia začína vo všetkých súčasne. Koncových stavov môže byť v diagrame prítomných viac kvôli viacerým možným priebehom. V prípade, že sa tok riadenia dostane do ľubovoľného z nich dôjde k ukončeniu vykonávania daného diagramu aktivity.

3.2.4. Akcie

Akcia (Action) je základná vykonateľná jednotka v aktivity diagramoch podobne ako aktivita. Vykonanie akcie reprezentuje isté transformácie alebo procesy v modelovanom systéme (napr. vytvorenie nového objektu, priradenie hodnoty atribútu, ...). Všetky akcie sú v UML preddefinované. Môžu mať vstupy a výstupy, ktoré sa nazývajú piny. Piny sú spojené s objektmi pomocou dátového toku. Reprezentujú prúd dát do, z akcie. Všetky používané akcie v diagramoch sú inštanciami tried metamodelu UML definovaného v špecifikácii UML. Akcie sú jediné elementy UML, ktoré môžu volať objekty, môžu meniť ich stav, vykonávať na nich operácie a dovolávať sa priamo užívateľsky definovaného správania (aktivít), či už priamo alebo prostredníctvom operácie. Z týchto príčin sú často nazývané primitívne dynamické elementy UML a taktiež z dôvodu, že každé správanie musí byť nakoniec zredukované do akcií, aby malo nejaký efekt na objektoch alebo sa mohlo dovolávať iného správania. Akcie nie sú sami o sebe správaním, pretože sú poskytnuté priamo v UML na rozdiel od aktivít, ktoré sú definované užívateľom. UML definuje dostatok akcií pre aplikácie rôzneho zamerania. Akcie sú priamo používané iba v diagramoch aktivity. Ostatné časti UML ich môžu používať iba skrze aktivity diagramy. Ukončenie akcie závisí na jej interných podmienkach. Po ukončení sú dáta predané do výstupných pinov a tok riadenia pokračuje v elementoch s ňom spojených.

3.2.5. Prechody

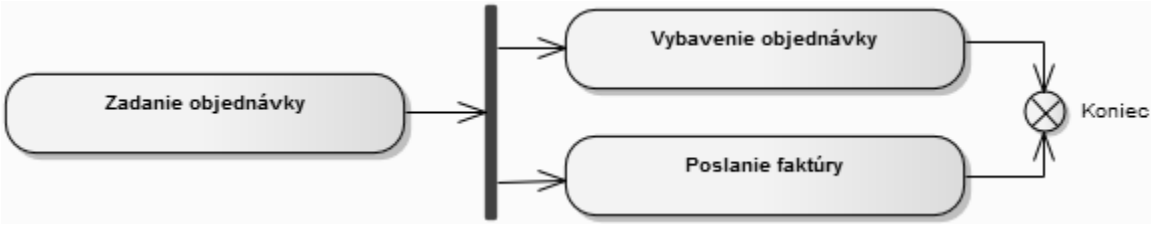
V UML rozlišujeme dva druhy prechodov (Flow transitions)

- Tok riadenia (Control flow).
- Dátový tok (Object flow).

Tok riadenia

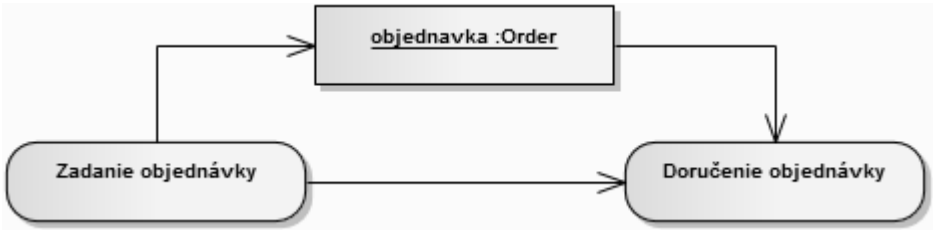
Slúži na uskutočnenie prechodu medzi jednotlivými elementmi. Ak vykonávanie daného elementu skončí, je riadenie predané nasledujúcim elementom, ktoré sú s ním spojené prechodmi. Tok riadenia je zobrazený prostredníctvom orientovanej šípky. Jeho vykonávanie sa začína v počiatočnom stave a prebieha pokým sa nedostane do niektorého z koncových stavov. Prostredníctvom toku riadenia je jednoznačne určené poradie vykonávania jednotlivých elementov. K hrane, ktorá ho reprezentuje, je možné pridať jej názov alebo stručný popis.

Tok riadenia je možné ukončiť pomocou špeciálneho elementu *Flow final*. Z tohto dôvodu tento element neobsahuje žiadny výstupný tok riadenia. Celková aktivita v grafe je ukončená potom ako sú ukončené všetky jej prechody.



Dátový tok

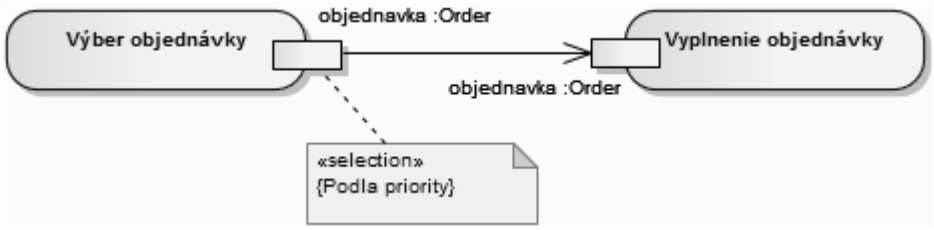
Slúži na prepojenie vstupov alebo výstupov elementov s objektmi. Typicky zobrazuje použitie objektov, ktoré sú dôležité pri vykonávaní činnosti popísanej daným diagramom aktivity alebo tok dát medzi jednotlivými elementmi. U akcií a aktivít je dátový tok typicky napojený na piny alebo parametre (ActivityParameterNodes), ale nie je to podmienka (vtedy sa jedná o tzv. implicitný pin/parameter). Na rozdiel od toku riadenia, dátový tok nie je možné duplikovať (napr. pomocou vetvenia). K prechodu, ktorý ho reprezentuje môže byť pridaný jeho názov alebo stručný popis. Taktiež môže mať nastavenú váhu pomocou notácie *{weight = n}*. Hodnota váhy (n) je kladné celé číslo (môže byť určené i konštantou) alebo hodnota *null* (často označovaná ako *all*). Účelom váhy je obmedzenie použitia daného prechodu. Jej presný význam závisí na konkrétnom použití a kontexte. Pre dátový tok bolo zavedené nasledujúce obmedzenie. Vstup elementu nemôže prijať vstupné dáta pokiaľ ich súčasne nemôžu prijať i jeho ostatné vstupy. Túto podmienku UML aplikuje z dôvodu predchádzania deadlocku, ktorý by mohol nastať v prípade, ak by vstupy dvoch konkurenčných elementov obsahovali vstupné dáta potrebné druhým z nich. V takom prípade, by sa nemohol vykonať ani jeden z nich, keďže žiaden nemá všetky potrebné vstupné dáta.



K dátovému toku je možné pridať poznámku s nasledujúcimi stereotypmi

- *Transformation* – popisuje transformáciu, ktorá sa uplatní na každom objekte, ktorý ním prechádza.

- *Selection* – definuje spôsob, akým budú dáta posielané týmto dátovým tokom napr. použitím prioritnej fronty.



Typy objektov dátového toku

- *Pin* – slúži na definovanie dátového vstupu alebo výstupu realizovaného príslušným elementom. Jeho bližší popis sa nachádza v samostatnej sekcii.
- *Activity parameter node* – slúži na definovanie dátového vstupu alebo výstupu realizovaného príslušnou aktivitou. Jeho význam je podobný vstupnému resp. výstupnému pinu.

- *Central buffer* – je objekt so stereotypom *<<centralBuffer>>*. Používa sa v situácii, v ktorej dáta prichádzajú od viacerých zdrojov súčasne a takisto môžu byť viacerým zdrojom poskytované.

- *Data store* - slúži na simulovanie vlastností dátového toku používaného pred zavedením UML 2.0, ktorý má nasledujúce vlastnosti

- *Je pasívny* - prítomnosť dát v data store neaktivuje vykonávanie napojených elementov. Tie si dáta vyžiadavajú podľa potreby.
- *Je nevyčerpateľný* - použitie dát obsiahnutých v data store nespôsobí ich odstránenie.
- *Je perzistentný* - dáta v data store ostávajú i po tom, ako element, ktorý ich vlastní, skonči svoje vykonávanie.

Element reprezentujúci objekt môže obsahovať nasledujúce popisy

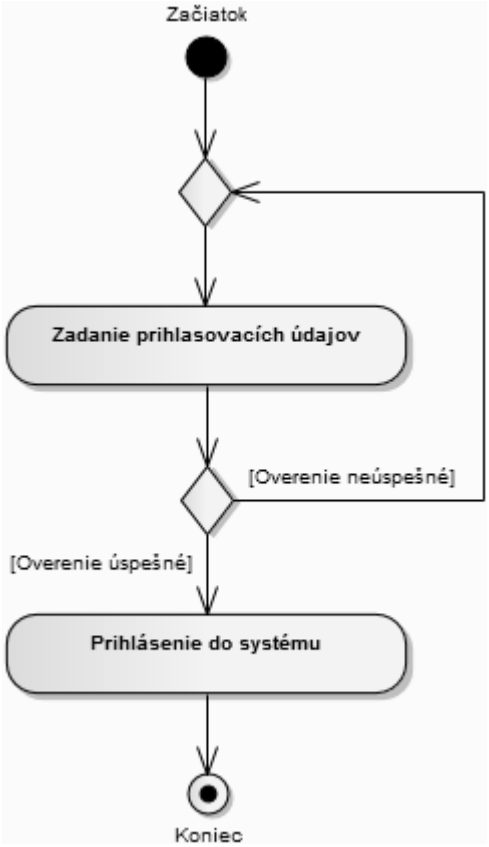
- *názovInštancie : Trieda* – inštancia triedy Trieda s názvom názovInštancie.
- *názovInštancie* – inštancia neznámej triedy s názvom názovInštancie.
- *: Trieda* – anonymná inštancia triedy Trieda.

K objektovým elementom je taktiež možné priradiť rôzne notácie definované špecifikáciou UML akou je napr. *{upperBound = 100}*, ktorej význam je podobný ako význam notácie *weight* u hrany dátového toku. Interpretácia týchto notácií nie je jasne definovaná a teda závisí od konkrétnej situácie a kontextu, v ktorom je použitá.

3.2.6. Rozhodnutia a zjednotenia

Rozhodnutie (Decision) slúži na realizáciu alternatívnych prechodov v diagrame. Prechod, ktorým sa vydá tok riadenia, je určený na základe logickej podmienky, ktorá je priradená k jednotlivým jeho výstupným hranám (guard expression). Podmienky sú zapísané v hranatých zátvorkách. Logické podmienky priradené k jednotlivým vetám musia byť vo vzájomnom vylúčení (Mutual exclusion), aby bolo zaistené, že daný diagram aktivity bude deterministický. Na jednej z výstupných vetví je možné použiť kľúčové slovo *[else]*. Vetva s touto podmienkou bude vykonaná v prípade, ak nebola splnená logická podmienka žiadnej z ostatných výstupných vetví. UML nedefinuje poradie vyhodnocovania podmienok na jednotlivých vetvách, teda môžu byť teoreticky vyhodnocované súčasne. Z toho vyplýva, že logické podmienky priradené k výstupným hranám by nemali mať vedľajšie efekty. Rozhodnutia majú iba jeden vstup, ktorého hodnota môže byť použitá pri vyhodnocovaní jednotlivých podmienok, a minimálne dva výstupy (alternatívne prechody).

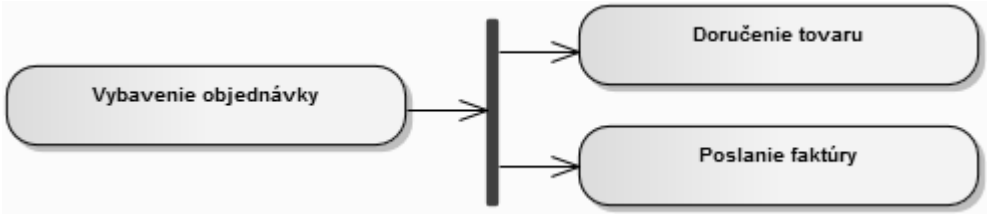
K rozhodovaciemu elementu je možné priradiť poznámku (note) so stereotypom *<<decisionInput>>*, ktorá obsahuje výpočet (napr. aktivitu), na ktorého výsledok sa môžu odkazovať jednotlivé vetvy pri definovaní logických podmienok.

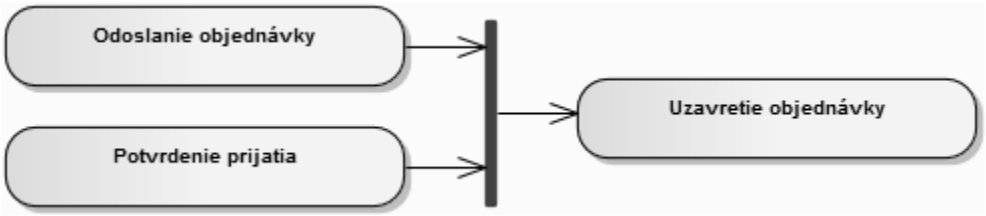


Opakom rozhodnutia je zjednotenie (Merge) viacerých prechodov do jedného. Pre jeho realizáciu je použitý rovnaký symbol, avšak v tomto prípade je na vstupe dva a viac prechodov a na výstupe práve jeden. Na výstupnú vetvu nesmie byť kladená žiadna logická podmienka.

3.2.7. Vetvenia a spojenia

Vetvenie (Fork) slúži na modelovanie procesov, ktoré bežia paralelne. Na vstupe majú práve jeden prechod, na výstupe dva a viac, ktoré sú na sebe navzájom nezávislé. Vykonávanie elementov napojených na výstupné hrany sa začína v jednom okamihu, teda paralelne.





Spojenie (Join) slúži na opätovné zjednotenie paralelných procesov. Na vstupe majú dva a viac prechodov, na výstupe práve jeden. Výstupný prechod je aktivovaný až potom, ako boli dokončené všetky aktivity spojené so vstupnými prechodmi. Počet prechodov, ktoré sú na výstupe niektorého z vetvení musí byť rovnaký, ako počet prechodov, ktoré vstupujú do odpovedajúceho spojenia.

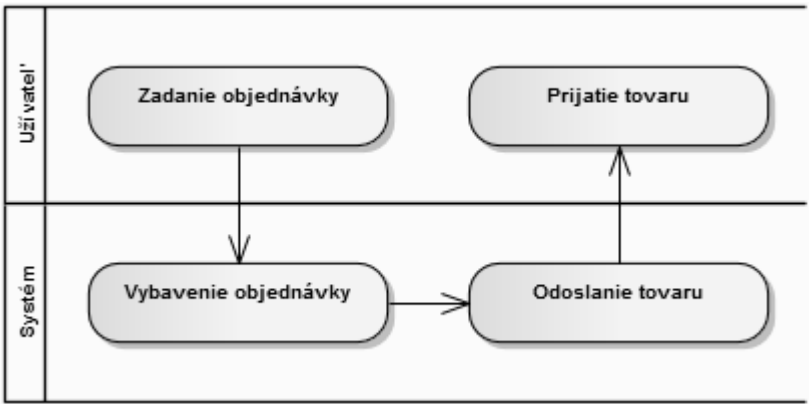
Spájanie prechodov sa riadi dvoma pravidlami

- Ak sú všetky vstupné prechody tokmi riadenia, tak výsledkom je jediný výstupný tok riadenia.
- Ak sú niektoré zo vstupných prechodov tokmi riadenia a iné dátovými tokmi, tak všetky dátové toky sú spojené a poslané na výstup, pričom všetky toky riadenia zaniknú.

K elementu je možné pridať podmienku, na základe ktorej sa spojenie uskutoční. Napr. je možné pomenovať jednotlivé vstupné prechody a na výstupe spojiť iba niektoré z nich. Táto podmienka sa zadáva prostredníctvom notácie *{joinSpec = podmienka napr. (A or B) and C}*.

3.2.8. Oblasti zodpovednosti

Oblasti zodpovednosti (Swimlanes) slúžia na rozdelenie diagramov aktivity do oblastí napr. pre lepši prehľad alebo pre ujasnenie toho, ktoré časti systému sú zodpovedné za vykonávanie určitých činností. Oblasti zodpovednosti nemajú žiadnu sémantiku. V prípade ich použitia musí každý z použitých elementov okrem prechodov náležať do práve jednej z nich. Prechody ako jediné môžu krížiť ich hranice v prípade, ak spájajú elementy patriace do rôznych oblastí zodpovedností.



Jednotlivým oblastiam zodpovednosti je možné priradiť nasledujúce stereotypy (uvedený popis slúži ako príklad ich realizácie v cieľovom jazyku)

- *Class* – v tomto prípade všetky elementy obsiahnuté v danej oblasti budú súčasťou triedy, ktorá ju reprezentuje vo forme metódy.
- *Property* – v tomto prípade je oblasť podriadená oblasti so stereotypom *class*. Reprezentuje určitú vlastnosť tejto triedy. Každá z vlastností je samostatnou triedou, pričom elementy obsiahnuté v danej oblasti sú reprezentované formou jej metódy.
- *Instance* – v tomto prípade je oblasť podriadená oblasti so stereotypom *class* alebo *property*. Elementy obsiahnuté v danej oblasti budú reprezentované formou metódy pomenovanej inštancie triedy, ktorá reprezentuje nadradenú oblasť.

Oblastiam je možné priradiť vlastné stereotypy a taktiež je možné ich modelovanie do viacerých dimenzií.

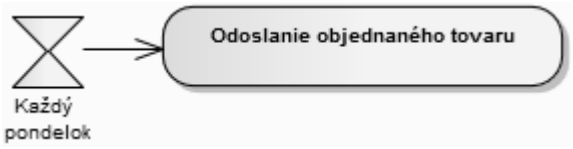
3.2.9. Signály

Signály (Signals) slúžia na asynchrónnu výmenu informácií v rámci diagramu aktivity. Pre odoslanie a príjem signálu slúžia dva rôzne elementy. Môžu byť napojené na tok riadenia podobne ako ostatné elementy. Nie je možné s nimi asociovať žiadnu činnosť, jediné čo vykonávajú je odoslanie alebo príjem signálu. Jeho vyslanie môže byť zviazané s konkrétnym objektom (napr. inštanciou triedy). V takom prípade je informáciou vyslanou signálom práve tento objekt.

Daný signál je možné použiť i ako stereotyp (*<<signal>>*). Interne je reprezentovaný podobne ako obyčajná trieda, avšak s tým rozdielom, že obsahuje iba atribúty a implicitnú operáciu send(targetSend) umožňujúcu poslanie signálu množine cieľových objektov. Signál môže mať svoje inštancie a taktiež smie byť použitý pri generalizácii. Podobne ako triedy, signály môžu mať atribúty a operácie. Atribúty signálu slúžia ako jeho parametre. Súčasťou špecifikácie každého elementu použitého v rámci diagramu aktivity by mal byť i zoznam signálov, ktoré môže svojimi operátormi posielať.

3.2.10. Časové udalosti

Časové udalosti (Time events) slúžia na vyvolanie udalostí v špecifický časový okamih alebo opakovane v zadaný časový interval (každý deň, týždeň, ...). Znázorňujú sa prostredníctvom symbolu presýpacích hodín. V prípade, že je časová udalosť napojená na tok riadenia, môže byť aktivovaná až potom, ako je spracovávaná. Teda daná udalosť je vyvolaná iba raz v určený čas. Ak má daná časová udalosť nastávať periodicky nesmie obsahovať žiadny vstup.

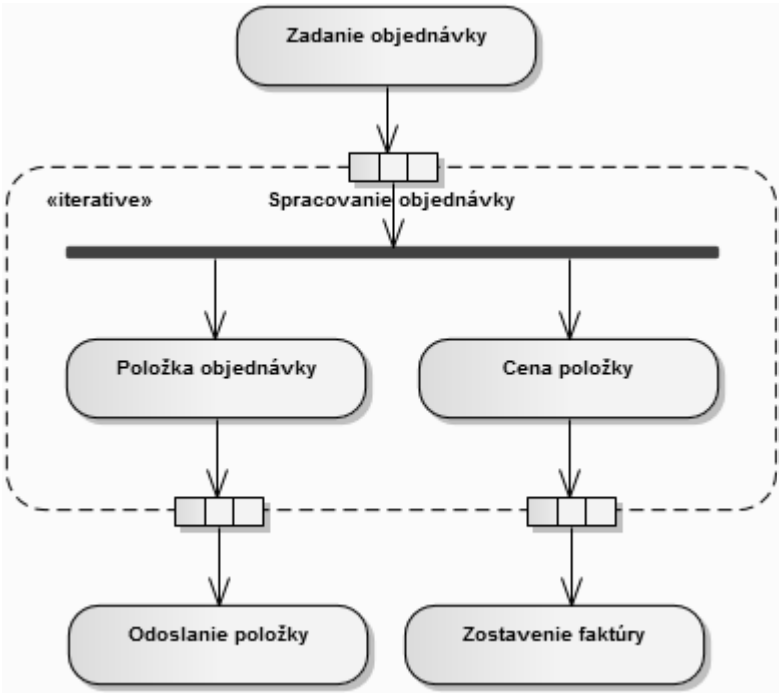


3.2.11. Rozširujúci región

Rozširujúci región (Expansion region) sa používa v prípade, ak má byť určitá postupnosť operácií uplatnená na každom prvku vstupnej množiny. Bez jeho použitia by bolo nutné danú situácia riešiť prostredníctvom modelovania cyklu. Vstupom i výstupom *expansion regionu* je množina elementov. Jeho obsahom je postupnosť operácií, ktoré budú aplikované. Na vstupe je množina rozložená na jednotlivé elementy a na každom z nich je vykonaný obsah regiónu. Po spracovaní elementov je výstup (ak existuje) umiestnený do výstupnej množiny, v ktorej sú elementy v rovnakom poradí ako na vstupe. Región má typicky jeden vstup a výstup. Všeobecne, ale môže mať jeden a viac vstupov a žiadny alebo viac výstupov. Všetky množiny (vstupné i výstupné) musia mať rovnakú veľkosť, ale nemusia obsahovať rovnaké typy hodnôt. Vstupy a výstupy regiónu sú reprezentované objektmi *expansion nodes*. Každá hodnota vstupujúca do regiónu mimo nich je braná ako konštanta. Týka sa to i vstupov realizovaných prostredníctvom pinov.

K expansion regionu môže byť priradený jeden z nasledujúcich stereotypov

- *Iterative* – jednotlivé prvky vstupnej množiny sú spracovávané postupne.
- *Paralell* – jednotlivé prvky vstupnej množiny sú spracovávané súčasne. Každý z prvkov je spracovaný v samostatnej kópii regiónu, pričom medzi jednotlivými prvkami nie je žiadna forma interakcie.
- *Stream* – jednotlivé prvky vstupnej množiny prichádzajú postupne a sú i postupne spracovávané.



3.2.12. Piny

Piny (Pins) reprezentujú dátové vstupy a výstupy elementov a umožňujú ich zapojenie do dátového toku. Uchovávajú vstupy, pokým sa element nezačne vykonávať a výstupy, pokým nie je riadenie predané ďalej. K pinu môže byť priradený názov, ktorý sa typicky používa na označenie typu dát, ktoré daný pin prijíma alebo poskytuje. Pin musí byť kompatibilný s typom dát, ktoré prichádzajú cez dátový tok. Ak nemá špecifikovanú hodnotu, tak pracuje s dátami ľubovoľného typu. K pinom môže byť priradený efekt, ktorý má daný element na vstupnom alebo výstupnom objekte, prostredníctvom notácie *{effect = ...}*.

Effect môže nadobúdať jednu zo 4 hodnôt

- *Create* – dáta boli vytvorené daným elementom. Táto hodnota môže byť použitá iba na výstupnom pine.
- *Read* – dáta sú používané len na čítanie.
- *Write* – dáta môžu byť modifikované.
- *Delete* – dáta budú zmazané daným elementom. Táto hodnota môže byť použitá iba na vstupnom pine.

UML definuje nasledujúce špeciálne typy pinov

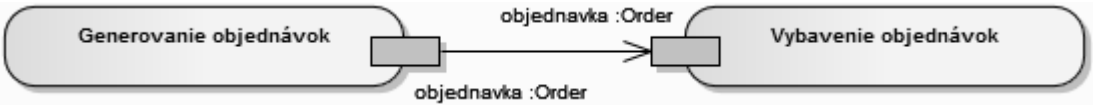
- Standalone pin.
- Stream pin.
- Exception pin.
- Value pin.

Standalone pin

Je pin vyskytujúci sa samostatne mimo elementov. V takom prípade stotožňuje ako vstupný tak i výstupný pin.

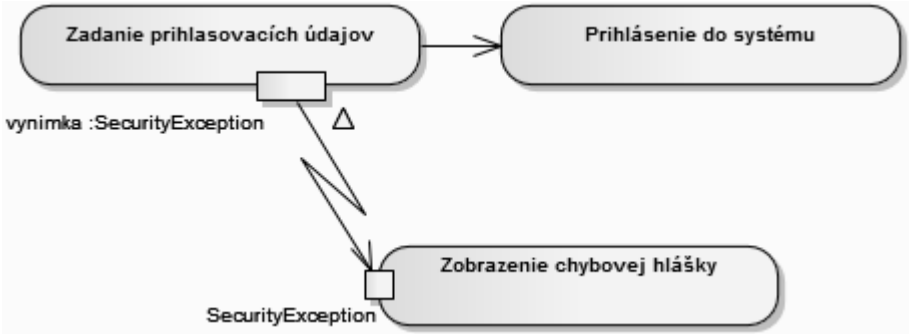
Stream pin

Používa sa v prípade, ak element na vstupe prijíma alebo na výstupe poskytuje viac hodnôt v priebehu svojho vykonávania. Znázorňuje sa pomocou notácie *{stream}* alebo pomocou pinu s čiernym pozadím.



Exception pin

Používa sa v prípade, ak je dátový výstup prepojený na spracovanie výnimky. Znázorňuje sa pomocou trojuholníka zobrazeného v jeho tesnej blízkosti. Po jeho vyvolaní nesmú byť poskytnuté ostatné výstupy ani zmena toku riadenia a vykonávanie aktivity musí byť okamžite prerušené. Element môže mať viac takýchto pinov, ale výstup môže byť vykonaný len na jednom z nich.



Value pin

Je špeciálnym typom vstupného pinu, ktorý poskytuje elementu konštantnú hodnotu. Tá býva uvedená pri jeho symbole. Typ hodnoty musí byť kompatibilný s typom vstupného pinu. Value pin nemôže byť použitý ako výstup elementu.

Ďalšie grafické notácie používané v súvislosti s pinmi

- Množiny pinov.
- Odlíšenie vstupných a výstupných parametrov.

Množiny pinov

Slúžia na rozdelenie vstupných alebo výstupných parametrov do logických skupín. Daný element môže prijímať vstupy a poskytovať výstupy iba v rámci jednej skupiny. Skupina musí obsahovať minimálne jeden pin, ale jeden pin môže byť obsiahnutý vo viacerých skupinách.

Odlíšenie vstupných a výstupných parametrov

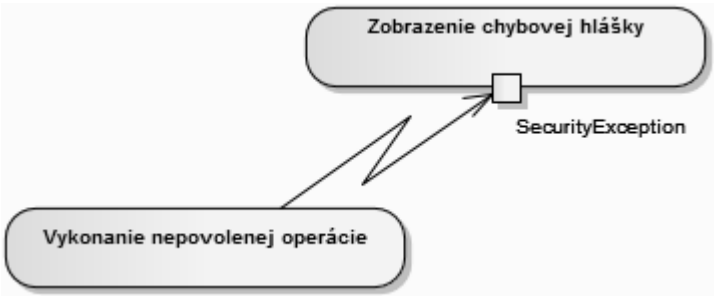
V prípade, že s daným elementom nie je združený žiadny prechod, je možné odlíšenie vstupných a výstupných pinov pomocou symbolu šípky umiestneného v ich stredoch. Vstupný pin obsahuje šípku smerujúcu do vnútra elementu, výstupný smerov von.

3.2.13. Výnimky

Výnimky (Exceptions) predstavujú v UML možnosť ako vyvolávať a obsluhovať výnimky, a teda i výnimočné stavy v modely, ktoré môžu nastať pri jeho vykonávaní. Samotné vyvolanie výnimky je znázornené prostredníctvom šípky vo forme blesku.

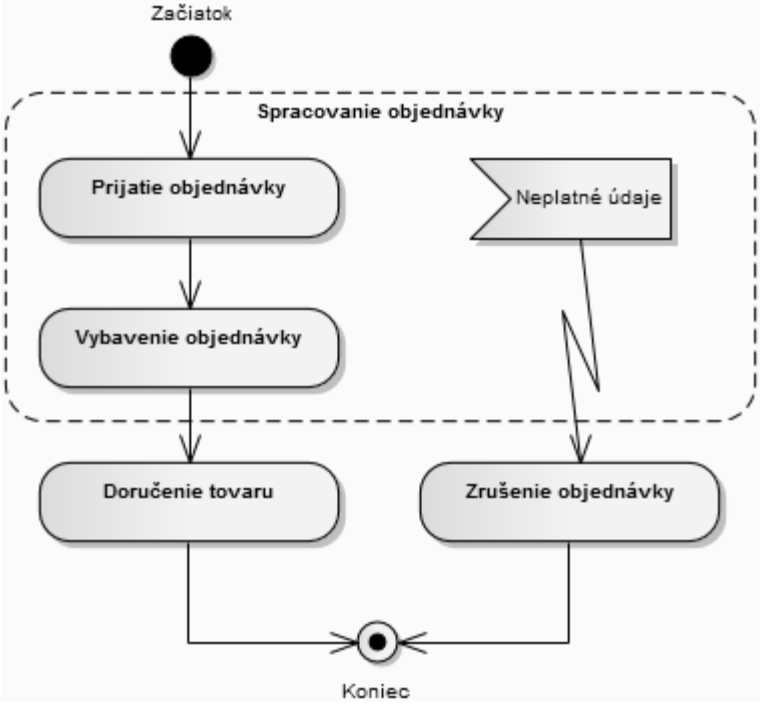
Odchytenie a spracovanie výnimky

Pre tento účel je používaný element *Exception handler*, ktorý definuje obsluhu vykonanú v prípade nastania výnimky pri spracovaní takzvaného chráneného elementu (zviazaného s exception handler prostredníctvom výnimky). Každý element môže byť prepojený s viacerými exception handlers v závislosti na tom, aké rôzne výnimky majú byť odchytávané a spracované. Po vyvolaní výnimky nastane zistenie, ktorý z handlerov je zodpovedný za jej ošetrenie. Ak taký existuje je vykonaný jeho obsah a následne sa pokračuje vo vykonávaný všetkých elementov spojených s chráneným elementom, ktorý danú výnimku vyvolal. V opačnom prípade je daná výnimka propagovaná do nadradeného bloku.



Vyvolanie výnimky z množiny elementov

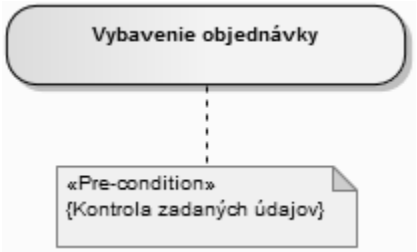
Pre tento účel slúži element *Interruptible Activity Region*. Ten obsahuje množinu navzájom prepojených elementov za účelom možnosti ukončenie toku riadenia prostredníctvom vyvolania výnimky z ľubovoľného z nich behom spracovania. V prípade, že vyvolaná výnimka prechádza hranice regiónu, dôjde k prerušeniu vykonávania všetkých elementov obsiahnutých v jeho vnútri. Samotná vyvolanie výnimky je realizovaná prostredníctvom poslania signálu.



3.2.14. Vstupné a výstupné podmienky

K jednotlivým elementom je možné pridávať vstupné (Preconditions) a výstupné (Postconditions) podmienky, u ktorých sa predpokladá, že sú splnené pred alebo po samotnom vykonaní elementu. Tieto podmienky sa pridávajú prostredníctvom poznámky (note) so stereotypmi

- *<<localPrecondition>>* - pre vstupnú podmienku.
- *<<localPostcondition>>* - pre výstupnú podmienku.



UML priamo nedefinuje kontrolu a vyhodnotenie týchto podmienok. Teda to, kedy a v akom kontexte sa majú dané podmienky validovať a takt isto neurčuje ako sa má ošetriť situácia v ktorej dôjde k porušeniu týchto podmienok pri vykonávaní určitého elementu.

4. XMI

Cieľom tejto kapitoly je popis formátu XMI, ktorý bude slúžiť ako vstup pre prototyp umožňujúci generovanie kódu. Na jeho úplný popis je nutné taktiež oboznámenie s formátom XML a štandardom MOF. Prostredníctvom formátu XML je zapísaný samotný obsah XMI. Ten odpovedá metamodelu, ktorý je možné vyjadriť prostredníctvom MOF. Kapitola je z tohto dôvodu rozdelená na nasledujúce logické časti

- Popis formátu XML.
- Popis formátu XMI. Zhodnotenie jeho výhod a nevýhod. Popis rozdielností medzi jeho jednotlivými verziami.
- Popis štandardu MOF.

4.1. XML

Extensible Markup Language (XML) je otvorený štandard konzorcia W3C slúžiaci na prenos a výmenu obecných dát (ako textov, tak i dátových štruktúr). Je podmnožinou SGML (Standard Generalized Markup Language), ktorý nie je rozšírene používaný práve kvôli svojej komplexnosti. Obsahuje dáta v textovej podobe so silnou podporou Unicode. Definuje množinu pravidiel, prostredníctvom ktorých je možné definovať význam jednotlivých častí dokumentu. Takáto reprezentácia má oproti iným formátom výhodu predovšetkým pri vyhľadávaní konkrétneho typu informácie. XML nedefinuje zobrazenie samotného dokumentu, je to predovšetkým syntaktický nástroj. Napriek tomu existujú viaceré štýlové jazyky slúžiace danému účelu (napr. jazyk XSL, ktorý navyše umožňuje dokument pred samotným zobrazením príslušne transformovať).

Štruktúra XML dokumentu pozostáva z nasledujúcich častí

- *Prolog* – obsahuje deklaráciu verzie XML, prípadne informáciu o použitom kódovaní.

```
<?xml version='1.1' encoding='iso-8859-2' ?>
```

- *Gramatika* – definuje logickú štruktúra dokumentu. Obsahuje deklarácie, zoznam povolených elementov, komentáre a inštrukcie pre spracovanie. Môže byť zapísaná napr. prostredníctvom DTD, XML Schema alebo jazyka Schematron.

- *Obsah* – obsahuje samotné reprezentované dáta, ktorých zápis odpovedá gramatike (ak je definovaná).

Obsah XML dokumentu je zložený z elementov (základných stavebných prvkov), ktoré sú do seba navzájom vnorené. Na najvyššej úrovni sa nachádza jediný element označovaný tiež ako koreňový, v rámci ktorého sú vnorené všetky ostatné. Je to jedna z podmienok validného XML dokumentu. Jednotlivé elementy sa zapisujú prostredníctvom začiatočného a koncového tagu, medzi ktorými je umiestnený ich obsah. Názov tagu sa uzatvára medzi znakmi < a >, pričom koncový tag má pre svoje odlíšenie navyše pred názvom uvedený znak /.

```
<name>Meno</name>
```

Príklad zápisu elementu *name*.

Výnimkou je tzv. prázdny element, ktorý nemá obsah. V takomto prípade je možné použiť skrátenú notáciu, v ktorej je vynechaný koncový tag a začiatočný je navyše ukončený znakom /.

```
<br/>
```

Príklad zápisu prázdneho elementu *br*.

Každý počiatočný tag v rámci XML dokumentu musí byť ukončený odpovedajúcim koncovým tagom (okrem skrátenej notácie prázdneho elementu), pričom jednotlivé uzátvorkovania sa navzájom nesmú krížiť. Opäť ide o jednu z podmienok, ktorú musí spĺňať validný XML dokument. U počiatočného tagu je typicky možné použitie atribútov v závislosti od konkrétneho elementu. Tie sú používané za účelom spresnenia jeho obsahu. Hodnotu atribútu je vždy nutné uzatvoriť do úvodzoviek alebo apostrofov. V prípade, ak má element viac atribútov sú od seba navzájom oddelené prostredníctvom medzier.

```
<img src='obrazok.jpg'/>
```

Príklad zápisu elementu *img* s atribútom *src*.

Ďalšie syntaktické prvky, ktoré je možné použiť v rámci XML dokumentu

- *Komentár* – je súčasťou dokumentu, ale je ignorovaný parserom. Slúži na vysvetlenie jednotlivých časti dokumentu, prípadne ich skrytie pred spracovaním. Zapisuje sa medzi postupnosti znakov <!--, -->.

```
<!--komentár -->
```

Príklad zápisu komentára v rámci XML dokumentu.

- *Sekcia CDATA* – typicky slúži na vloženie väčšieho množstva textu, ktorý môže obsahovať ľubovoľné znaky, teda napr. i <, >. Tento text totiž nie je spracovaný parserom, ale na rozdiel od komentára je súčasťou výstupu.

```
<![CDATA[
Tento text nebude
spracovaný parserom
]]>
```

Príklad zápisu sekcie CDATA.

- *Inštrukcie pre spracovanie* – umožňujú vloženie riadiacich informácii do dokumentu. Tie slúžia programom, ktorým je daný dokument určený pre spracovanie. Ide o dvojice obsahujúce názov inštrukcie a dáta s ktorými pracuje. Zapisujú sa prostredníctvom nasledujúcej notácie.

```
<? Inštrukcia dáta ?>
```

Notácia pre zápis inštrukcie.

4.2. XMI

XML Metadata Interchange (XMI) je otvorený štandard Object Management Group (OMG) pre výmenu metadát prostredníctvom eXtensible Markup Language (XML). Jeho hlavným účelom je umožnenie jednoduchej výmeny metadát predovšetkým medzi CASE nástrojmi, ale všeobecne je možné jeho použitie na výmenu ľubovoľných metadát, ktorých metamodel môže byť vyjadrený prostredníctvom Meta-Object Facility (MOF). Keďže XMI slúži na uloženie UML modelu prostredníctvom formátu XML, je jeho súčasťou taktiež sada DTD, ktorým exporty CASE nástrojov musia odpovedať.

Príklady

- *Počiatočný uzol*

```
<node xmi:type="uml:InitialNode"
xmi:id="EAID_628165F3_601A_4c3e_8322_E5B503FCB1C3" name="ActivityInitial"
visibility="public"/>
```

Je reprezentovaný prostredníctvom elementu *node* s atribútom *xmi:type* hodnoty *uml:InitialNode*. Atribút *xmi:id* obsahuje jedinečný identifikátor uzlu, *name* obsahuje názov uzlu a *visibility* obsahuje dostupnosť uzlu v rámci aktivity, ktorého je súčasťou.

- *Vstupná hrana elementu*

```
<incoming xmi:idref="EAID_BE31E687_5113_429c_92AF_F90E555D84E2"/>
```

Je reprezentovaná prostredníctvom elementu *incoming*. Atribút *xmi:idref* obsahuje odkaz na definíciu hrany. Ak chceme vyjadriť, že daná hrana je vstupná pre určitý uzol napr. vyššie definovaný počiatočný uzol, musíme ju uviesť ako podelement daného uzlu.

```
<node xmi:type="uml:ActivityFinalNode"
xmi:id="EAID_7821D07C_51CF_4d7c_A3BB_1B7A40B9A71C" name="ActivityFinal"
visibility="public">
      <incoming xmi:idref="EAID_BE31E687_5113_429c_92AF_F90E555D84E2"/>
</node>
```

- *Podriadený element*

Vyjadrenie, že určitý uzol je súčasťou nadradeného uzla sa v XMI reprezentuje všeobecne prostredníctvom podelementov. Napríklad situáciu, v ktorej je akcia obsiahnutá v rámci štruktúrovanej aktivity, je možné vyjadriť nasledovne.

```
<group xmi:type="uml:StructuredActivityNode"
xmi:id="EAID_ADC9621B_62FC_4360_B3C0_4D37A1A0EABC" name="Structured activity"
visibility="public">
      <containedNode xmi:type="uml:CallBehaviorAction"
      xmi:id="EAID_AE50E96A_3B0A_4271_874E_72AD6C6450EA" name="Action"
      visibility="public"/>
</group>
```

4.3. Výhody

Požívanie len grafického zobrazenia UML modelu nie je typicky dostačujúce. Napríklad za účelom analýzy konzistencie modelu alebo generovania kódu je vhodné použitie externého nástroja v prípade, ak modelovací nástroj takúto funkcionalitu neposkytuje. Vstup v takejto situácii musí byť v štandardizovanom formáte, keďže môže pochádzať z ľubovoľného CASE editora. XMI je štandard prijatý samotnými výrobcami. Poskytuje vývojárom pracujúcim s objektovými technológiami prostriedky pre výmenu dát prostredníctvom internetu v štandardizovanom tvare. Kombinuje výhody štandardu XML slúžiaceho pre definovanie, validáciu a zdieľanie dokumentov a výhody objektovo orientovaného UML. Vďaka používaniu

dát v štandardizovanom formáte XMI je možná výmena dát medzi jednotlivými nástrojmi, aplikáciami a repozitármi. Takisto je možné vytváranie distribuovaných aplikácií. Ďalšou nespornou výhodou je, že XMI používa na výmenu dát formát XML. Na spracovanie XML existuje v súčasnosti množstvo nástrojov a takmer každý vyšší programovací jazyk obsahuje pre jeho spracovanie natívnu podporu napr. vo forme knižníc.

Príklad

Technológia .NET poskytuje implementáciu StAX párseru (XMLTextReader, XMLTextWriter) a DOM párseru (XMLNodeReader, XMLNodeWriter).

4.4. Nevýhody

Formát XMI neposkytuje prostriedky ako zaznamenať rozmiestnenie a vzhľad jednotlivých elementov UML diagramu. Tieto informácie budú po exportovaní do XMI navždy stratené, a preto jednotlivé elementy po ich importovaní editorom musia byť znova rozmiestnené. Niektoré editory umožňujú pridanie týchto informácií, avšak tie nie sú súčasťou štandardu, a teda sú typicky podporované len daným editorom. Ďalšou nevýhodou tohto formátu je, že pre malé modely môže byť vygenerovaný relatívne veľký XMI súbor. Dôsledkom round-trip pri exportovaní UML modelu do XMI formátu je používanie odkazov na jednotlivé elementy, ktoré sa vyskytujú v modely viackrát alebo je nutné sa na nich odkázať viac ako raz. Klasickým príkladom je zápis hrán.

```
<edge xmi:type="uml:ControlFlow"
xmi:id="EAID_79AC697F_3A7B_4083_AF55_2E5450062344" visibility="public"
source="EAID_BE9552D2_CEF4_4bc9_A558_8543DD30B2F9"
target="EAID_F38628B4_D402_4a0f_9F8D_EBE2A4B7232D"/>
```

Elementy, ktoré hrana spája sú určené prostredníctvom atribútov *source* a *target*, ktoré obsahujú ich identifikátory. Odkazy na elementy môžu byť použité pred ich samotným definovaním, keďže špecifikácia XMI nedefinuje poradie v akom sa majú vyskytovať. Teda po spracovaní XMI súboru a jeho prevedení do vlastnej reprezentácie je nutné ešte samotné prepojenie elementov na základe ich identifikátorov.

4.5. Verzie

Novo kladené požiadavky na výmenu dát si vynútili taktiež početné zmeny vo formáte XMI. Tie boli postupne zavádzané v jednotlivých jeho verziách, ktoré sú od seba odlíšené prostredníctvom sprievodného čísla. Aktuálne existuje štandard XMI v nasledujúcich verziách : 1.0, 1.1, 1.2, 2.0 a 2.1. Najpočetnejšie rozdiely sa vyskytujú medzi verziami začínajúcimi číslicami 1 a 2. Najväčším problémom týchto zmien je fakt, že sa nejedná iba o inkrementálne rozširovanie funkcionality, pričom tá stávajúca ostáva bez zmien. Dôsledkom toho je nekompatibilita vedúca k tomu, že výstup v XMI verzie 1.x nie je možné spracovať nástrojom, ktorý obsahuje iba podporu verzie 2.0 a vyššej. Z tohto dôvodu sa v tejto diplomovej práci obmedzíme iba na použitie XMI vo verzii 2.1, keďže tá má podporu pre UML 2.0, ktorého nové vlastnosti sú v tejto diplomovej práci potrebné pri generovaní kódu.

Nasledujúca tabuľka obsahuje zhrnutie hlavných rozdielov medzi verziami XMI 1.0 a 2.1.

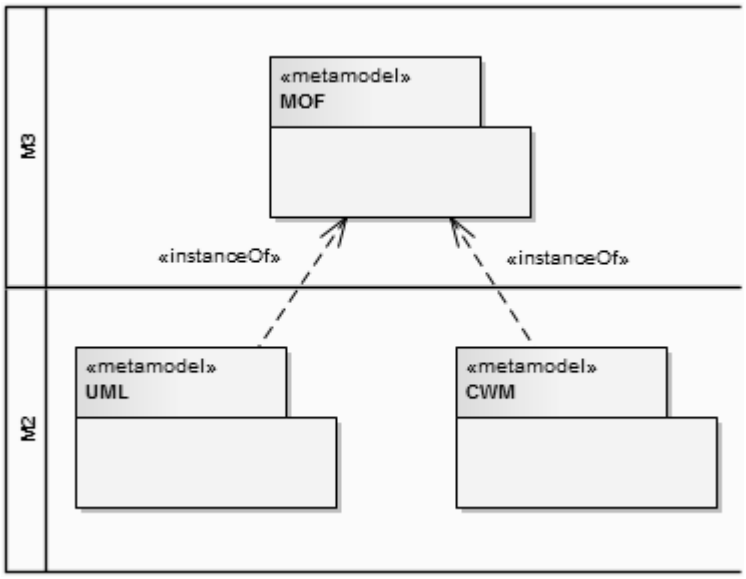
Vlastnosť	*Verzia XMI 1.0*	*Verzia XMI 2.1*
Serializácia	Nepodporovaná	Podporovaná
Názov elementov	Používa sa plne kvalifikovaný názov elementu napr. *Foundation.Core.Class*	Používa sa iba názov elementu napr. *UML:Class*
Identifikácia elementov	Prostredníctvom xmi.id a xmi.uuid	Prostredníctvom xmi.id, xmi.label, xmi.uuid
Prepojenie elementov	Prostredníctvom xmi.id	Prostredníctvom xmi:idref
Identifikácia XML	Nepodporovaná	Prostredníctvom xml.id
Generovanie do XMI Schema (XSD)	Nepodporované	Podporované
Metamodel	Odkaz na metamodel použitý v dokumente uložený v XMI.metamodel	Každý metamodel definuje 1 a viac namespaceov, prostredníctvom ktorých je dostupný
Podpora MOF, UML	MOF 1.3, UML 1.3	MOF 2.0, UML 2.0, UML 2.1
Podpora	Podporované takmer všetkými nástrojmi	Podporované zvyčajne iba profesionálnymi nástrojmi

4.6. MOF

Meta Object Facility (MOF) je štandard Object Management Group (OMG) slúžiaci na vytváranie a správu metamodelov. Obsahuje služby umožňujúce vývoj a spoluprácu jednotlivých modelov ako i systémov riadených metadátami. Architektúra MOF sa skladá zo 4 vrstiev (M3, M2, M1 a M0).

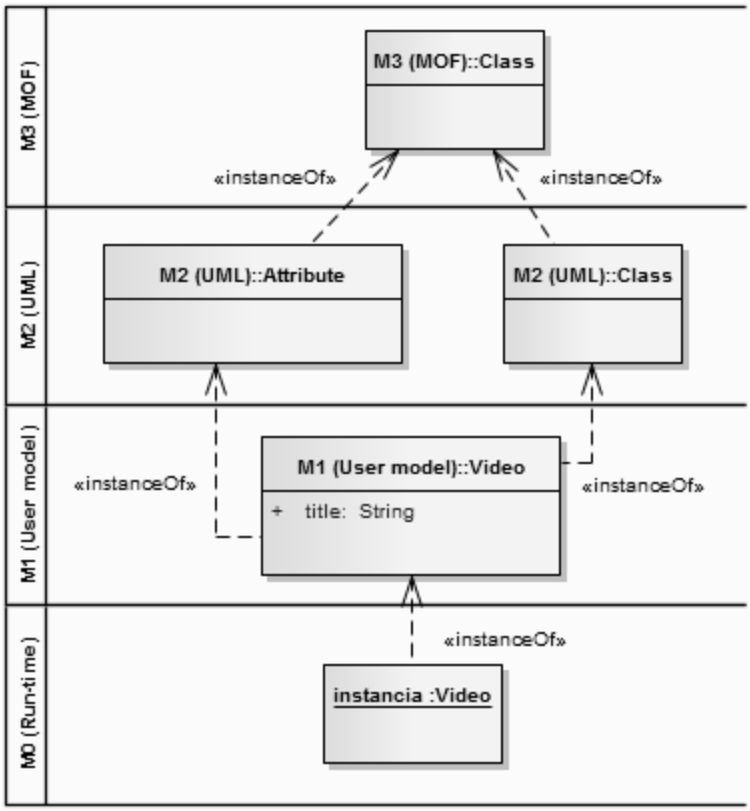
Popis jednotlivých vrstiev

- M3 – obsahuje meta-meta model umožňujúci definovanie a vytváranie metamodelov na druhej úrovni.
- M2 – obsahuje metamodely definované prostredníctvom MOF. Tieto metamodely definujú elementy a ich možné prepojenia pri vytváraní konkrétnych modelov na prvej úrovni. Medzi ich najznámejších predstaviteľov patrí UML a CWM



- M1 – obsahuje modely vytvorené použitím elementov definovaných metamodelom druhej úrovne. V prípade UML takýmto modelom môže byť napríklad konkrétny diagram aktivity.
- M0 – označovaná taktiež ako dátová vrstva. Na rozdiel od prvej vrstvy sa v modely vyskytujú už konkrétne inštancie elementov, ktoré sa používajú pri samotnom vykonávaní modelu.

Každý element vyskytujúci sa na vrstve rôznej od M3 je asociovaný s odpovedajúcim elementom nadradenej vrstvy.



V tejto diplomovej práci sa budeme opierať výlučne o metamodel UML definovaný na vrstve M2. Teda každá zmienka o metamodely v nasledujúcom texte sa bude odkazovať práve na tento metamodel, ak nebude bližšie špecifikovaný.

5. Generovanie kódu

5.1. Úvod

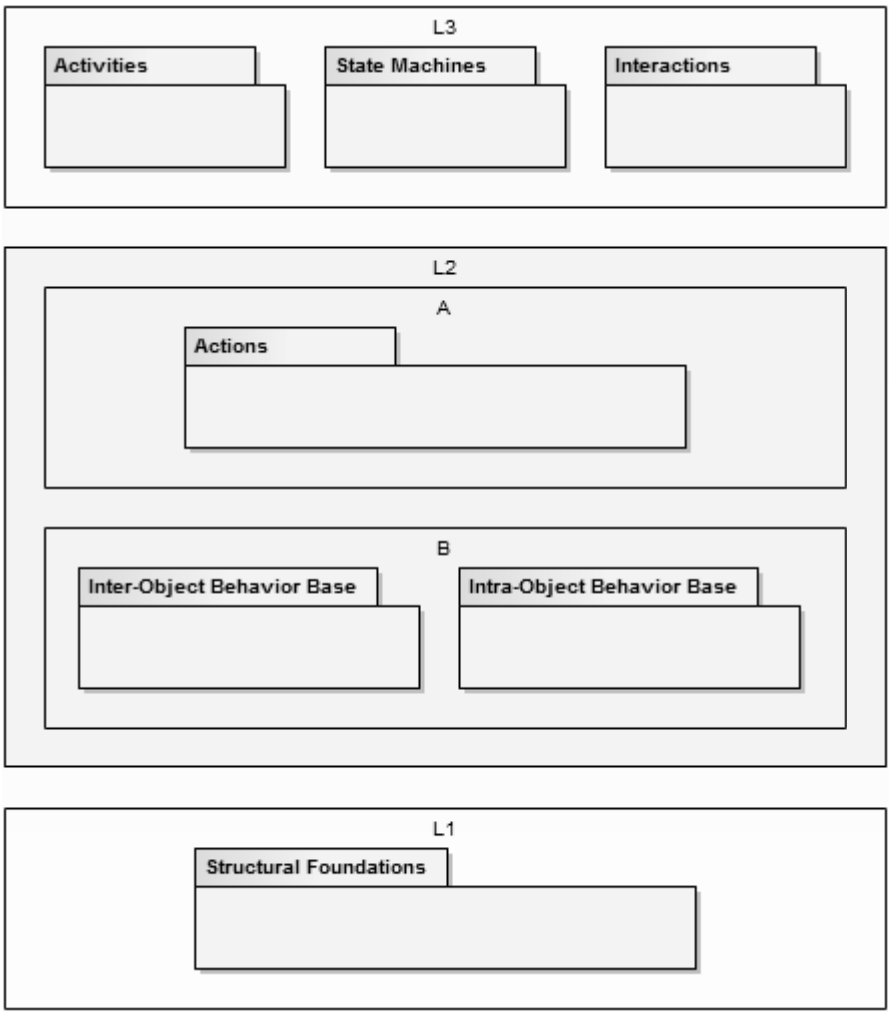
UML je prevažne vnímaný ako prostriedok umožňujúci znázornenie modelovaných systémov prostredníctvom grafickej notácie. Táto reprezentácia je síce dominantnou, pretože väčšinou postačuje, avšak pre niektoré účely je nevhodnou. Takýmto prípadom je napríklad situácia, v ktorej je nutné z modelovaného systému vygenerovať kostru kódu vykonateľného na konkrétnej cieľovej platforme alebo v konkrétnom cieľovom jazyku. Grafická reprezentácie je v tomto prípade nevhodná predovšetkým z hľadiska sémantiky, keďže tá nie je jednoznačne určená a môže sa v závislosti od kontextu meniť. Tejto problematike je v rámci špecifikácie UML venovaná časť s názvom „*Run-time semantics*". Jej hlavným cieľom je definovanie exekučného prostredia (execution environment) a mapovania modelovaných situácii do reprezentácie vhodnej na ich samotné vykonanie (simulovanie modelu). Táto reprezentácia je súčasťou UML metamodelu (viz. kapitola MOF) a je označovaná ako repository. V nasledujúcich častiach textu bude pod týmto názvom taktiež často referencovaná. Je nutné si uvedomiť, že nie každý element modelovaného systému je možné reprezentovať ako postupnosť príkazov programovacieho jazyka (Poznámky, elementy umožňujúce štruktúrovanie diagramu do logicky súvisiaci oblastí, ...). Z tohto dôvodu zavádza UML dva predpoklady, vďaka ktorým je možné generovanie kódu z elementov modelujúcich správanie.

- Každé správanie v modelovanom systéme je vyvolané prostredníctvom odpovedajúcej akcie.
- Sémantika sa uplatňuje iba na udalosťami riadené elementy diagramov správania.

5.2. Architektúra sémantického modelu

Popisuje jednotlivé oblasti sémantiky pokryté aktuálnym štandardom a to, ako sú navzájom prepojené. Tieto oblasti je možné znázorniť ako horizontálne vrstvy, pričom prvky obsiahnuté na niektorej z nadradených vrstiev závisia od prvkov vrstvy podradenej.

Pri vysokej abstrakcii môžeme architektúru rozdeliť do nasledujúcich troch vrstiev.



- L1 vrstva - zaručuje, že každý element diagramu správania obsahuje telo, teda je možné ho reprezentovať ako postupnosť akcií.
- L2 vrstva - je možné ju rozdeliť na dve podvrstvy. Vrstva B pozostáva z *Inter-Object Behavior Base*, ktorá definuje ako jednotlivé entity navzájom komunikujú a *Intra-Object Behavior Base* definujúcej správanie v rámci jednotlivých entít. Vrstva A definuje sémantiku jednotlivých akcií (Actions). Akcia je základná vykonateľná jednotka správania v rámci UML a je používaná na definovanie elementov správania na najnižšej úrovni detailu. Granularita vykonanej práce a vyjadrovacia sila akcií je porovnateľná s inštrukciami programovacieho jazyka. Podobne ako inštrukcie i akcie je možné rozdeliť do nasledujúcich kategórií v závislosti na type vykonávanej funkcionality
  - *Communication actions* - CallBehaviourAction, SendSignalAction, …
  - *Primitive function actions.*
  - *Object actions* - CreateObjectAction, DestroyObjectAction, …

- *Structural feature actions* - AddStructuralFeatureAction, ReadStructuralFeatureAction, …
- *Link actions* - CreateLinkAction, DestroyLinkAction, …
- *Variable actions* - ReadVariableAction, WriteVariableAction, …
- *Exception actions* – RaiseExceptionAction.

Vrstva L2 teda definuje reprezentáciu (Repository), ktorú je možné použiť ako medzikód v procese generovania kódu alebo vykonávania modelu.

- L3 vrstva - obsahuje jednotlivé elementy používané v diagramoch modelujúcich správanie, teda diagramoch aktivity, stavových diagramov a diagramov interakcie. Tieto elementy je možné formalizovať a popísať prostredníctvom nižšie položenej vrstvy obsahujúcej akcie.

5.3. Repository

Z vyššie popísaného textu vyplýva, že repository je interpretáciou diagramov správania prostredníctvom akcií, ako základných a ďalej nedeliteľných stavebných prvkov. Slúži ako prostredník medzi grafickou reprezentáciou modelovaného systému a jeho reprezentáciou v cieľovom jazyku. Každý objekt tejto reprezentácie je inštanciou metatriedy definovanej v rámci metamodelu UML. (viď. kapitola MOF). Jeho hlavným prínosom je odstránenie dvojzmyselnosti medzi jednotlivými interpretáciami modelovaných situácií a umožnenie kontroly konzistencie medzi nimi. Použitie tých istých elementov môže mať totiž rôznu sémantiku, a teda i iný význam v závislosti na kontexte. Repository umožňuje ich odlišnú reprezentáciu, a teda výsledný vygenerovaný kód bude odrážať presný úmysel analytika. CASE nástrojmi môže byť používaný ako cieľový jazyk, do ktorého sa má daný model skompilovať, čím sa zaistí konštrukcia iba konzistentných modelov. Z hľadiska generovania kódu plní účel tzv. medzikódu, ktorý je možné vzhľadom na svoje vlastnosti použiť i ako abstraktný syntaktický strom (výsledok parsera). Následne pri jeho prechode vhodným grafovým algoritmom je možné zostaviť cieľový kód.

5.4. Prevod diagramu aktivity do repository

V nasledujúcom texte popíšeme reprezentáciu niektorých elementov a situácií v rámci diagramu aktivity prostredníctvom repository.

5.4.1. Počiatočný uzol

Zobrazenie v UML.

Reprezentácia v repository pomocou metatriedy *InitialNode*.

5.4.2. Koncový uzol

Zobrazenie v UML.

Reprezentácia v repository pomocou metatriedy *ActivityFinalNode*.

5.4.3. Koniec toku riadenia

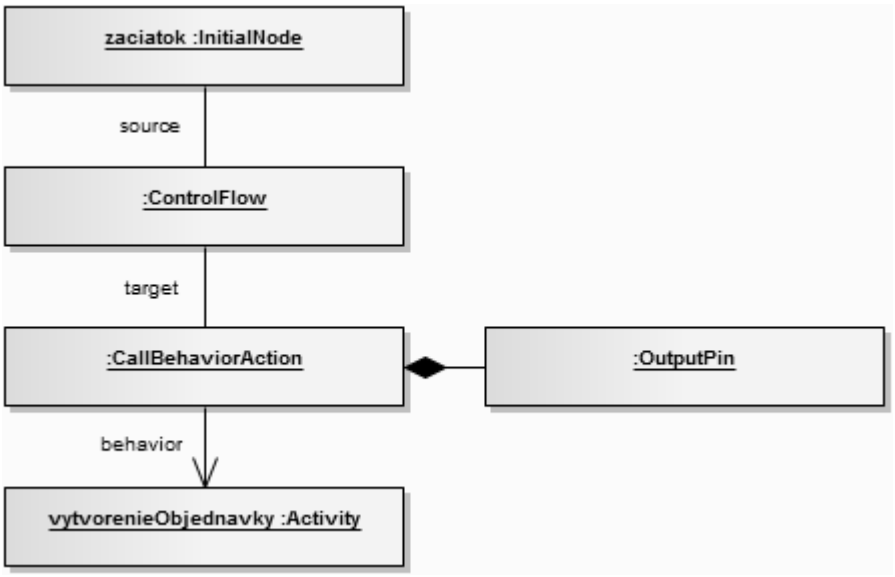
Zobrazenie v UML.

Reprezentácia v repository pomocou metatriedy *FlowFinalNode*.

5.4.4. Aktivita

Zobrazenie v UML.

Reprezentácia v repository pomocou metatried *InitialNode*, *ControlFlow*, *CallBehaviorAction*, *Activity* a *OutputPin*.



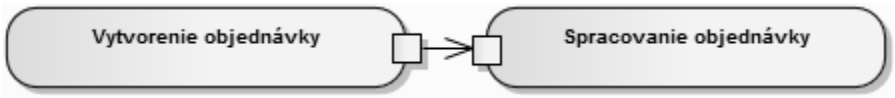
Význam metatried v danej situácii

- *InitialNode* - reprezentuje počiatočný stav.
- *ControlFlow* - reprezentuje tok riadenia medzi počiatočným stavom a aktivitou.

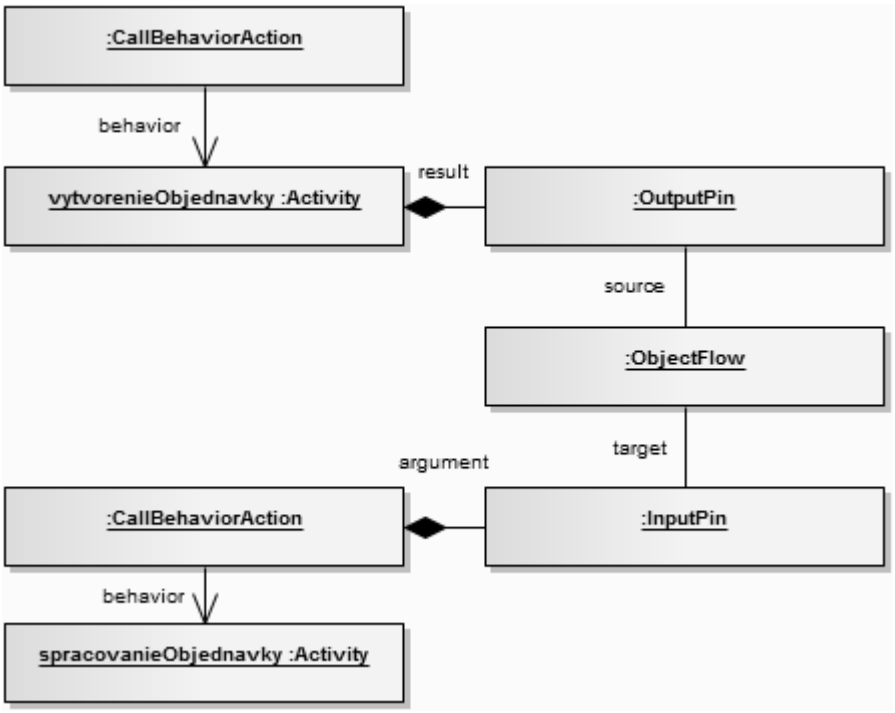
- *CallBehaviorAction* – reprezentuje vykonanie aktivity.
- *Activity* – reprezentuje samotnú aktivitu vytvorenia objednávky.
- *OutputPin* – reprezentuje výstup aktivity.

5.4.5. Prepojenie aktivít

Zobrazenie v UML.



Reprezentácia v repository pomocou metatried *CallBehaviorAction*, *Activity*, *ObjectFlow*, *OutputPin* a *InputPin*.



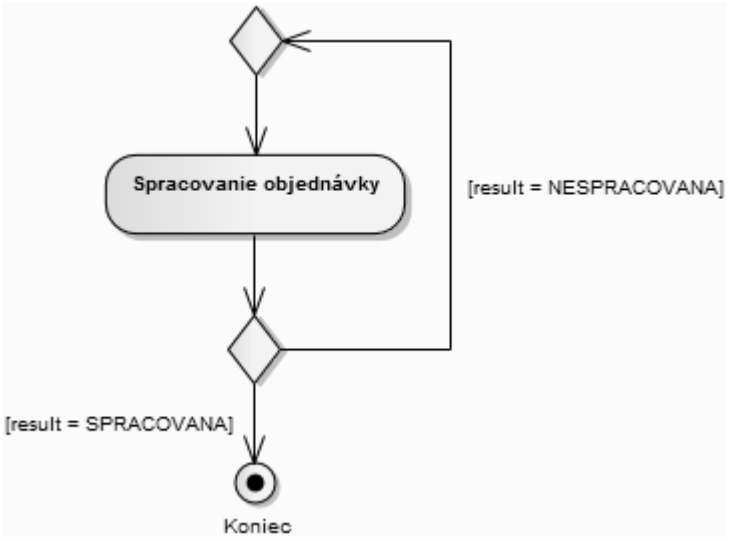
Význam metatried v danej situácii

- *CallBehaviorAction* – reprezentuje vykonanie aktivity.
- *Activity* – reprezentuje samotnú aktivitu. Inštancia *vytvorenieObjednavky* reprezentuje aktivitu vytvorenia objednávky a inštancia *spracovanieObjednavky* reprezentuje aktivitu spracovania objednávky.
- *ObjectFlow* – reprezentuje dátový tok medzi vstupom a výstupom aktivít.
- *OutputPin* – reprezentuje návratovú hodnotu aktivity pre vytvorenie objednávky.

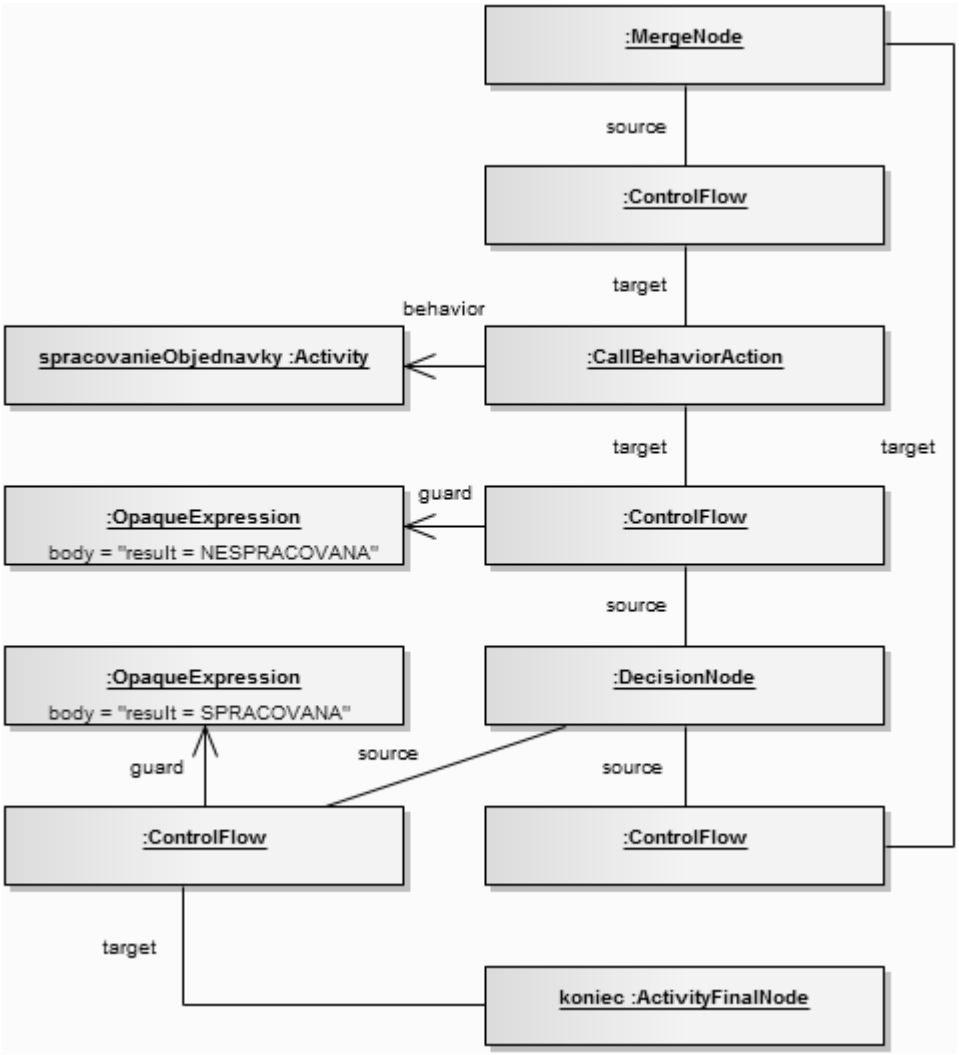
- *InputPin* – reprezentuje vstupný parameter aktivity pre spracovanie objednávky.

5.4.6. Cyklus

Zobrazenie v UML.



Reprezentácia v repository pomocou metatried *MergeNode*, *ControlFlow*, *CallBehaviorAction*, *DecisionNode*, *ActivityFinalNode*, *Activity* a *OpaqueExpression*.



Význam metatried v danej situácii

- *MergeNode* – reprezentuje koniec cyklu.
- *ControlFlow* – reprezentuje tok riadenia medzi jednotlivými objektmi.
- *CallBehaviorAction* - reprezentuje vykonanie aktivity.
- *DecisionNode* – reprezentuje začiatok vykonávania cyklu.
- *ActivityFinalNode* – reprezentuje koniec diagramu aktivity.
- *Activity* – reprezentuje samotnú aktivitu. Inštancia spracovanieObjednavky reprezentuje aktivitu spracovania objednávky.
- *OpaqueExpression* – reprezentuje podmienku, ktorá musí byť splnená, aby bolo možné pokračovať vo vykonávaní danou vetvou toku riadenia.

5.4.7. Prijatie signálu

Zobrazenie v UML.

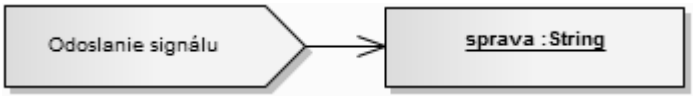
Reprezentácia v repository pomocou metatried *AcceptEventAction*, *Trigger*, *SignalEvent* a *Signal*.

Význam metatried v danej situácii

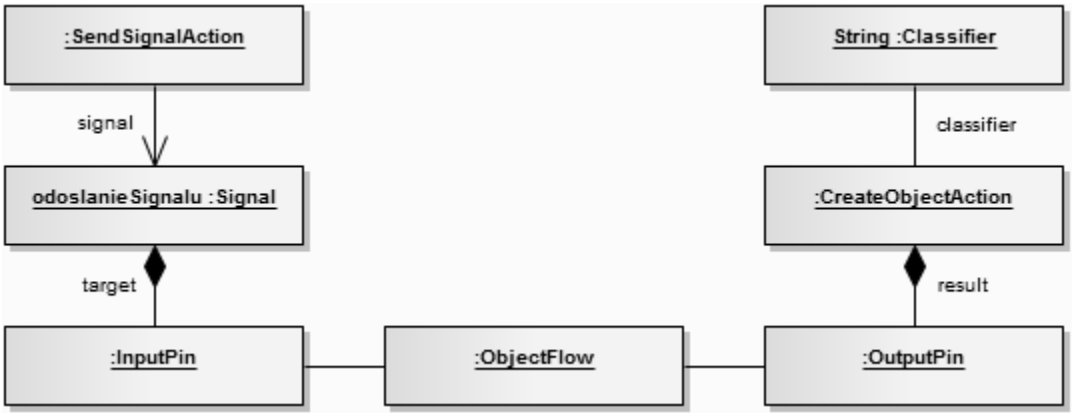
- *AcceptEventAction* – reprezentuje spracovanie udalosti.
- *Trigger* – reprezentuje typ spracovanej udalosti.
- *SignalEvent* – reprezentuje udalosť prijatia signálu.
- *Signal* – reprezentuje konkrétny prijatý signál.

5.4.8. Odoslanie signálu

Zobrazenie v UML.



Reprezentácia v repository pomocou metatried *SendSignalAction*, *Signal*, *InputPin*, *ObjectFlow*, *OutputPin*, *CreateObjectAction* a *Classifier*.



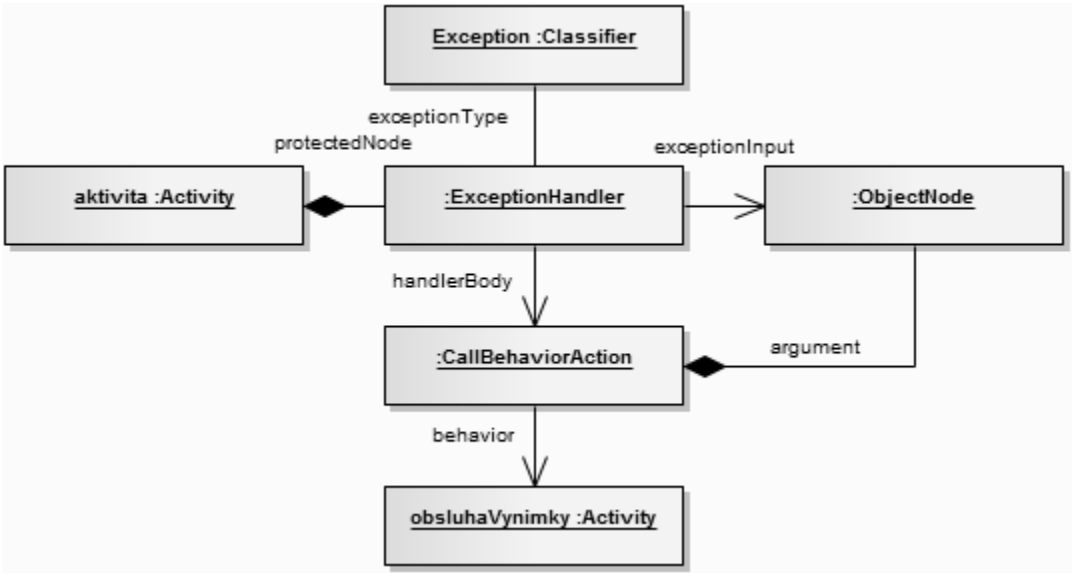
Význam metatried v danej situácii

- *SendSignalAction* – reprezentuje odoslanie signálu.
- *Signal* – reprezentuje konkrétny odosielaný signál.
- *InputPin* – reprezentuje vstupné dáta signálu.
- *ObjectFlow* – reprezentuje dátový tok.
- *OutputPin* – reprezentuje vytvorený objekt akciou *CreateObjectAction*.
- *CreateObjectAction* – reprezentuje vytvorenie nového objektu typu *String*.
- *Classifier* – reprezentuje typ vytváraného objektu akciou *CreateObjectAction*.

5.4.9. Vyvolanie výnimky z aktivity

Zobrazenie v UML.

Reprezentácia v repository pomocou metatried *ExceptionHandler*, *Activity*, *CallBehaviorAction*, *ObjectNode* a *Classifier*.

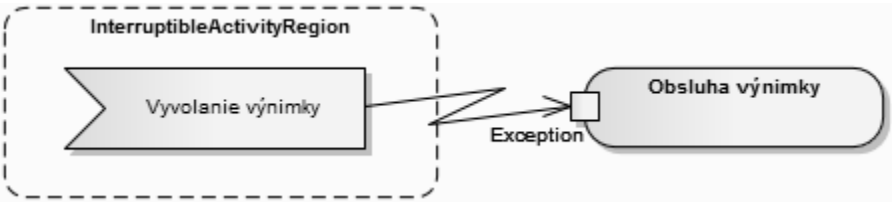


Význam metatried v danej situácii

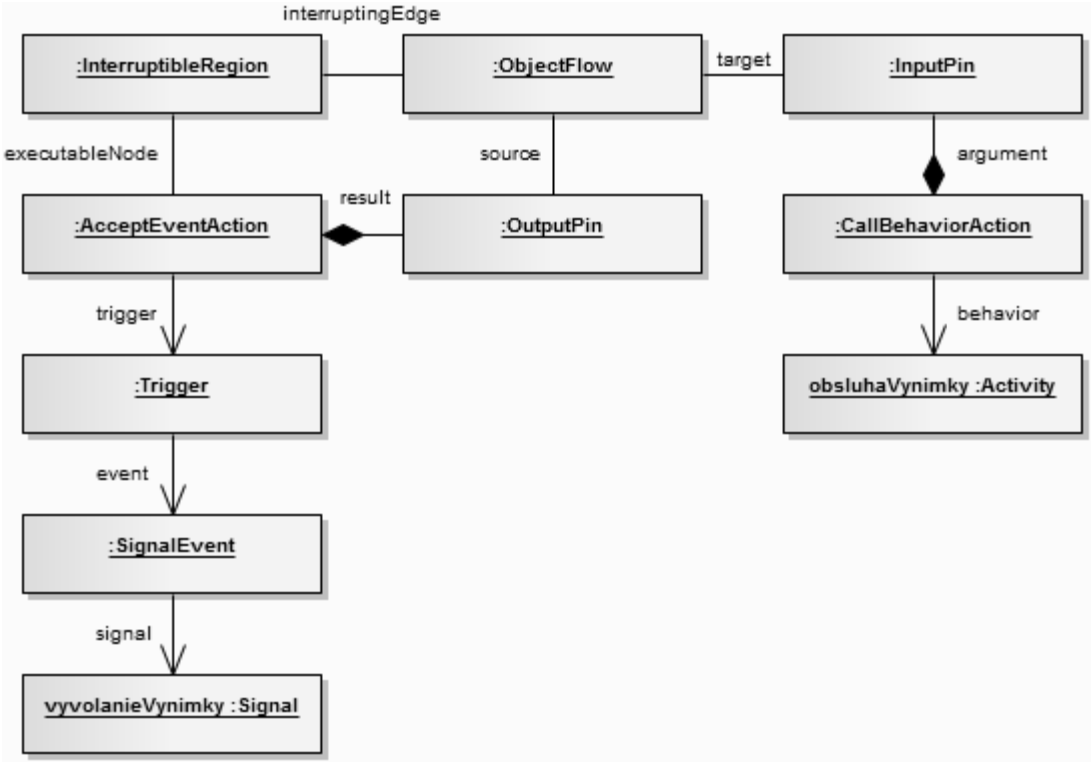
- *ExceptionHandler* – reprezentuje odoslanie signálu.
- *Activity* – reprezentuje samotnú aktivitu. Inštancia *obsluhaVynimky* je vykonaná po vyvolaní výnimky. Inštancia *aktivita* je tou, pri ktorej vykonávaní môže dôjsť k jej vyvolaniu.
- *CallBehaviorAction* – reprezentuje vyvolanie aktivity určenej na obsluhu výnimky.
- *ObjectNode* – reprezentuje argumenty vyvolanej výnimky.
- *Classifier* – reprezentuje typ odchytávanej výnimky.

5.4.10. Vyvolanie výnimky z prerušiteľného regiónu

Zobrazenie v UML.



Reprezentácia v repository pomocou metatried *InterruptibleRegion*, *AcceptEventAction*, *Trigger*, *SignalEvent*, *Signal*, *ObjectFlow*, *OutputPin*, *InputPin*, *CallBehaviorAction* a *Activity*.



Význam metatried v danej situácii

- *InterruptibleRegion* – reprezentuje chránenú oblasť, z ktorej sú odchytávané výnimky.
- *AcceptEventAction* – reprezentuje spracovanie udalosti.
- *Trigger* – reprezentuje typ spracovanej udalosti.
- *SignalEvent* – reprezentuje udalosť prijatia signálu.
- *Signal* – reprezentuje konkrétny prijatý signál.
- *ObjectFlow* – reprezentuje dátový tok.
- *OutputPin* – reprezentuje argumenty vyvolanej výnimky.
- *InputPin* – reprezentuje argumenty vyvolanej výnimky. Tie zároveň slúžia ako vstup aktivity majúcej na starosti jej obsluhu.
- *CallBehaviorAction* – reprezentuje vyvolanie aktivity určenej na obsluhu výnimky.
- *Activity* – reprezentuje aktivitu určenú na obsluhu vyvolanej výnimky.

6. Vlastné generovanie kódu

Cieľom tejto kapitoly je popis metódy generovania kódu použitej pri implementácii prototypu, ako i zvolenej architektúry za daným účelom. Kapitola je rozdelená na nasledujúce časti

- *Použitá metóda* – popisuje spôsob, ktorým je diagram aktivity prevedený zo vstupnej reprezentácie vyjadrenej prostredníctvom formátu XMI do postupnosti príkazov cieľového programovacie jazyka.
- *Architektúra* – popisuje rozčlenenie aplikácie na jednotlivé komponenty, ich význam a vzájomné prepojenie.
- *Štruktúra projektu* – obsahuje popis jednotlivých adresárov projektu obsahujúceho zdrojové kódy prototypu.
- *Modularita aplikácie* – popisuje spôsob, akým bola v rámci prototypu zaistená modularita, teda možnosť pridania modulov za účelom generovania kódu do nových cieľových jazykov.
- *Popis pomocných tried* – obsahuje popis pomocných tried, ktoré je možné využiť pri implementácii nového modulu.
- *Definícia nového elementu* – popisuje postup začlenenia nového elementu do procesu generovania kódu.

6.1. Použitá metóda

Cieľom tejto sekcie je popis metódy použitej pri generovaní kódu. Samotná metóda sa skladá z nasledujúcich krokov

- Extrahovanie jednotlivých elementov zo vstupného XMI súboru a ich následné prevedenie do reprezentácie definovanej prostredníctvom repository. Jednotlivé metatriedy definované v repository boli nahradené konkrétnymi triedami v jazyku C#. Tie boli vytvorené na základe špecifikácie UML. Keďže metamodel UML definuje rozsiahly objektový model, bola pre potreby diplomovej práce implementovaná iba podmnožina potrebná pre uloženie diagramu aktivity. Výsledkom tohto kroku je zostavenie grafovej reprezentácie diagramu, slúžiacej ako medzikód medzi grafickou reprezentáciou a reprezentáciou v cieľovom jazyku.

- Na takto skonštruovanej reprezentácii sú následne spočítané doplňujúce informácie uľahčujúce generovanie kódu ako napr. príznak, či hrana toku riadenia leží na cykle a pod.. Tieto informácie sú uložené v triedach reprezentujúcich dané objekty (čím došlo k rozšíreniu metatried definovaných v repository o ďalšie atribúty). Zároveň dochádza k pozmeneniu grafu za účelom jeho konzistentnosti. Takouto zmenou je napríklad pridanie vstupných a výstupných parametrov v prípade, ak sú dve aktivity prepojené priamo prostredníctvom dátového toku.
- Posledným krokom je aplikácia grafového algoritmu, ktorý využíva vyššie skonštruovaný graf ako abstraktný syntaktický strom. Pri jeho prechode sú na každom spracovanom vrchole grafu vyvolané príslušné udalosti generujúce cieľový kód.

6.1.1. Grafový algoritmus

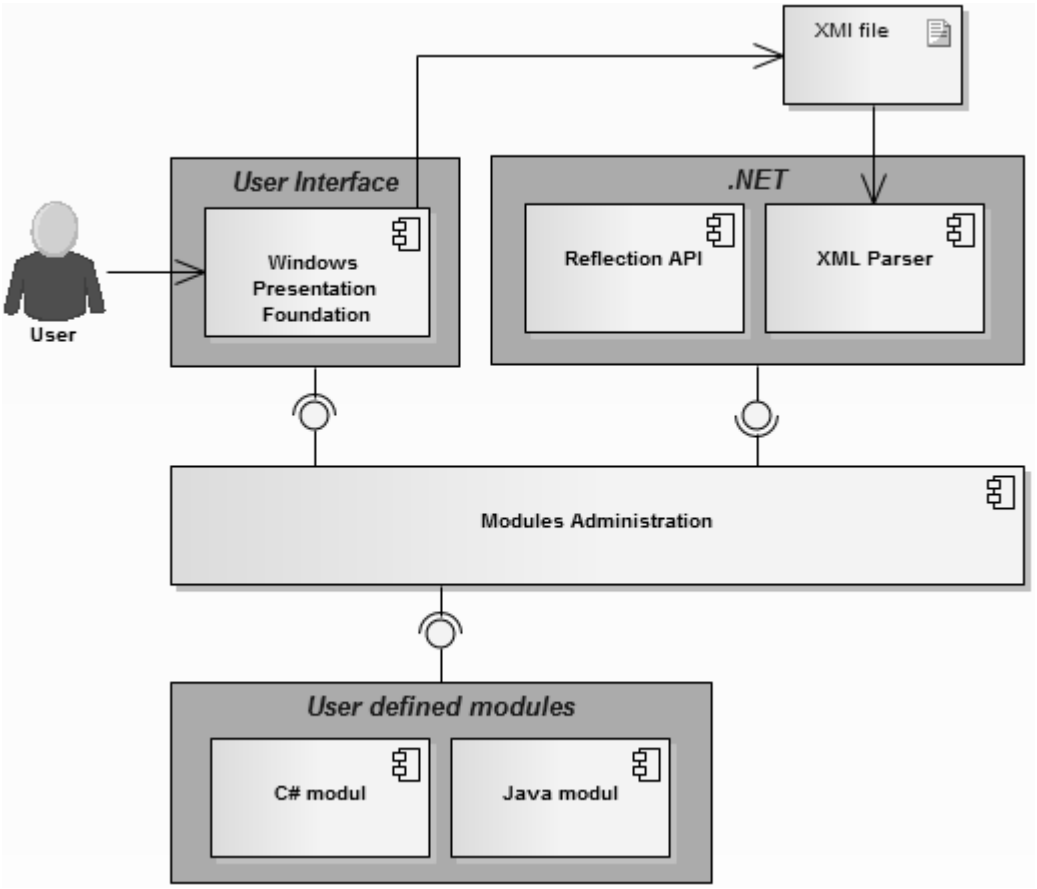
Grafový algoritmus použitý pri generovaní kódu má nasledujúce vlastnosti

- Vrcholmi grafu sú všetky elementu diagramu aktivity pomimo dátových objektov, ako napr. piny, vstupné a výstupné parametre aktivít, inštancie objektov a pod.. Hranami používanými k prechodu sú jedine hrany reprezentujúce tok riadenia. Dátové objekty a hrany dátového toku sa využívajú len pri generovaní vstupných a výstupných parametrov, predávania objektov, prípadne pri overovaní, či sú naplnené všetky dátové vstupy elementu pred jeho samotným spracovaním.
- Prehľadávanie grafu začína vždy z počiatočného uzla. V prípade generovania zo štruktúrovanej aktivity sa proces začína buď z počiatočného uzla alebo z elementov priamo napojených na jej vstupné parametre.
- Z elementu je generovaný kód (uskutočnený prechod) iba v prípade, ak má aktivované všetky vstupy v rámci toku riadenia a naplnené všetky dátové vstupy. Výnimkou je *merge node*, ktorý na svoju aktiváciu potrebuje iba jeden riadiaci vstup. Za účelom overenia naplnenosti dátových vstupov sú u jednotlivých elementov evidované typy a názvy ich výstupných objektov.
- Z každého elementu je generovaný kód iba raz. Výnimkou je pustenie algoritmu s príznakom lokálneho behu. Táto situácia je vhodná napríklad v prípade viacnásobného odkazovania na tú istú obsluhu výnimky.
- Algoritmus umožňuje vyvolanie definovaných udalostí na generovanie kódu v rôznych situáciách, akými napr. sú
    - Na začiatku spracovania elementu.
    - Na konci spracovania elementu.
    - Pred každým prechodom tokom riadenia vychádzajúceho z elementu.

- Po každom prechode tokom riadenia vychádzajúceho z elementu.
- ... (úplný popis udalostí viď. *„Rozhranie IModul"*).

6.2. Architektúra

Cieľom tejto sekcie je popis architektúry prototypu nástroja umožňujúceho generovanie kódu, ktorý je súčasťou diplomovej práce. Pre vývoj aplikácie bola zvolená technológia .NET a použitý programovací jazyk C#. Nasledujúci diagram popisuje prepojenie jednotlivých častí aplikácie ako i využívanie súčastí .NET framework.



Prototyp pozostáva z nasledujúcich častí

- *Užívateľské rozhranie* – je definované prostredníctvom Windows presentation foundation.
- *Knižnice .NET framework* – sú používané za účelom lokalizácie všetkých užívateľom definovaných modulov a za účelom spracovania vstupného XMI súboru.

- *Správa modulov* – zabezpečuje prepojenie jednotlivých častí prototypu, spracovanie vstupného XMI súboru, jeho prevedenie do reprezentácie definovanej prostredníctvom repository a aplikáciu samotného grafového algoritmu v rámci užívateľom zvoleného modulu.
- *Užívateľom definované moduly* – poskytujú API umožňujúce pridávanie modulov za účelom generovania kódu do nových cieľových jazykov.

Nasleduje popis jednotlivých častí.

6.2.1. Užívateľské rozhranie

Je vytvorené prostredníctvom WPF (Windows presentation foundation). Umožňuje užívateľovi výber modulu, ktorý bude použitý pri transformácii vstupného diagramu aktivity do cieľového jazyka, zobrazenie informácií o aplikácii, ako i konfiguráciu niektorých nastavení. Prostredníctvom definovaného rozhrania je možné v rámci UI získať zoznam všetkých užívateľom definovaných modulov, nastaviť cestu k súboru XMI, vyvolať samotné generovanie kódu, ako i predať referenciu na komponentu, v rámci ktorej budú zobrazované výstupné hlášky aplikácie. Samotné užívateľské rozhranie vystupuje v rámci aplikácie ako samostatná časť, a teda je možné ju zameniť bez akýchkoľvek zásahov do ostatných častí aplikácie. Generátor kódu je možné pozmeniť ako na webovú aplikáciu, tak i na konzolovú utilitu.

6.2.2. .NET Framework

Z technológie .NET boli použité predovšetkým nasledujúce súčasti

- *Reflection API* – prostredníctvom ktorého sú lokalizované užívateľom definované moduly v rámci projektu. Zároveň je použitý pre vytvorenie inštancie daného modulu za behu aplikácie a jej použitie pri generovaní kódu.
- *XML* – z .NET XML je použitý StAX parser za účelom analýzy vstupného XMI (popis viď. kapitola *„XMI"*) súboru obsahujúceho definície pre jednotlivé elementy diagramu aktivity, ktorý je predmetom generovania kódu.

6.2.3. Správa modulov

Obsahuje triedy zabezpečujúce komunikáciu medzi jednotlivými časťami aplikácie. Cez definované API umožňuje ovládanie aplikácie prostredníctvom užívateľského rozhrania. Zabezpečuje lokalizáciu a evidenciu všetkých užívateľom definovaných modulov a ich použitie pri generovaní kódu. Má na starosti spracovanie predloženého XMI dokumentu vrátane jeho transformácie do objektovej reprezentácie definovanej prostredníctvom repository, z ktorej je následne vytvorený abstraktný syntaktický strom. Ďalej je zodpovedná za vygenerovanie kódu z jednotlivých elementov stromu pri jeho postupnom prechádzaní podľa navrhnutého algoritmu.

6.2.4. Užívateľom definované moduly

Táto časť obsahuje API, cez ktoré je možné rozšíriť aplikáciu o ďalšie cieľové programovacie jazyky prostredníctvom pridávania nových modulov. V rámci diplomovej práce bol vyvinutý modul pre programovací jazyk C#.

6.3. Štruktúra projektu

Cieľom tejto sekcie je popis jednotlivých adresárov projektu *Dimplomova praca*, obsahujúceho zdrojové kódy prototypu. Projekt obsahuje nasledujúcu adresárovú štruktúru

- *Core* – obsahuje triedy definujúce jednotlivé elementy diagramu aktivity, ktoré sú spracovávané a je možné z nich generovať kód.
- *Moduly* – obsahuje triedy zabezpečujúce celkovú administráciu, teda prepojenie jednotlivých častí aplikácie do funkčného celku, akými sú užívateľské rozhranie, spracovanie XMI, vytvorenie abstraktného syntaktického stromu, lokalizácia užívateľom definovaných modulov a pod. Je ďalej členený na podpriečinky obsahujúce triedy pre jednotlivé moduly
    - *C#* - obsahuje triedy realizujúce generovanie kódu do programovacieho jazyka C#.
    - *Java* - obsahuje triedy realizujúce generovanie kódu do programovacieho jazyka Java. Modul aktuálne negeneruje žiaden kód. Bol vytvorený iba za účelom prezentovania modularity aplikácie.

- *Pomocné triedy* – obsahuje pomocné triedy, ktoré môžu byť využité programátorom modulov ako napr. triedy umožňujúce zobrazenie výstupov pri generovaní kódu vrátane chybových hlášok alebo triedy pre správu názvov (generovanie jednoznačných názvov, evidencia návratových typov a objektov metód pri ich volaní).

Bližší popis jednotlivých tried, ich metód a atribútov je obsiahnutý v rámci programátorskej dokumentácie, ktorá je súčasťou dodaného DVD.

6.4. Modularita aplikácie

Modularita aplikácie je zabezpečená prostredníctvom použitia udalosťami riadenej architektúry (Event-driven architecture). To je v programovacom jazyku C# možné zrealizovať prostredníctvom delegátov a eventov. V rámci modulu sú definované obsluhy udalostí, ktoré sú následne volané pri generovaní kódu. Tento prístup umožňuje obsluhu iba zvolených udalostí, a teda i generovanie kódu iba z požadovaných elementov. V rámci prototypu existujú nasledujúce typy udalostí, ktoré je možné rozdeliť do dvoch skupín

Udalosti vyvolávané pri generovaní kódu z elementu

- Generovanie kódu na začiatku spracovania elementu.
- Generovanie kódu na konci spracovania elementu.
- Generovanie kódu na začiatku prechodu hranou vychádzajúcou z elementu.
- Generovanie kódu z hrany, ktorou sa aktuálne prechádza.
- Generovanie kódu po uskutočnení prechodu hranou vychádzajúcou z elementu.
- Výskyt elementu pri prechode abstraktným syntaktickým stromom.

Udalosti vyvolávané pri generovaní kódu z diagramu aktivity ako celku

- Generovanie kódu na začiatku spracovanie aktivity diagramu.
- Generovanie kódu na konci spracovanie aktivity diagramu.
- Generovanie kódu z počiatočného uzlu.
- Generovanie kódu z koncového uzla.
- Načítanie XML elementu pri spracovaní vstupného XMI súboru.

Delegáty pre všetky tieto udalosti sú definované v rámci rozhrania *IModul*.

Nasleduje popis procesu pridania nového modulu, ktorý sa skladá z nasledujúcich častí

- Pridanie triedy reprezentujúcej nový modul.
- Implementácia udalostí v rámci rozhrania *IModul*.

6.4.1. Pridanie nového modulu

Nové moduly sa pridávajú programovo, ako triedy, v rámci projektu *"Diplomova praca"*. Sú umiestnené v príslušných podriečinkoch priečinka *Moduly*, ktoré sú po nich pomenované. Každý z nich obsahuje triedu reprezentujúcu daný modul, ako i pomocné triedy s ním súvisiace. Pre pomenovanie triedy obsahujúcej modul bola zavedená notácia *[Názov programovacieho jazyka]Modul.cs*. (Nie je nutné, ale vhodné dodržiavať). Modul je v rámci aplikácie realizovaný ako trieda implementujúca rozhranie *IModul*. Môže mať ľubovoľné umiestnenie v projekte a môže niesť ľubovoľný názov. Jedinou podmienkou je jej umiestnenie v rámci menného priestoru *Diplomova_praca*. Modul nie je nutné nikde registrovať ani vytvárať inštanciu príslušnej triedy. Jeho rozpoznanie a vytvorenie inštancie použitej pri generovaní kódu zaistí samotná aplikácia prostredníctvom reflection API.

6.4.2. Rozhranie IModul

Definuje rozhranie medzi samotným modulom a zvyškom aplikácie, prostredníctvom ktorého sú prístupné informácie o module a obsluhy jednotlivých udalostí. Dáta dostupné z modulu môžeme rozdeliť do troch kategórií

- *Informácie* – napr. vrátenie názvu modulu, ktorý bude užívateľovi zobrazený pri výbere v rámci UI.
- *Obsluha udalosti* – vrátenie odkazu na implementáciu metódy, ktorá bude použitá pri obsluhe danej udalosti.
- *Zoznam obslúh udalosti rôznymi elementmi* – vrátenie zoznamu dvojíc typ elementu a obsluha udalosti (odkaz na implementáciu metódy) pre konkrétnu udalosť. Obsluhy udalostí nastavajúcich pri generovaní kódu z jednotlivých elementov sa predávajú hromadne.

Príklad

Definovania obsluhy udalosti generovania kódu na začiatku spracovania elementu pre počiatočný uzol.

```
// InitialNode
public void GenerujKodZaciatok_InitialNode(Element element)
{
      InitialNode initialNode = element as InitialNode;
      Vystup.ZobrazRetazec("// " + initialNode.VratName());
}
```

Výsledkom metódy je zobrazenie názvu počiatočného uzla vo forme komentára na výstup. Dôležité je, aby rozhranie metódy odpovedalo príslušnému delegátovi pre danú udalosť. V tomto prípade je daný delegát definovaný v rámci rozhrania *IModul* nasledovne.

```
public delegate void GenerujKod(Element element);
```

Na obsluhu udalosti môže byť teda použitá iba metóda bez návratového typu prijímajúca jeden argument typu `Element`. Ten je následne nutné v rámci metódy pretypovať na požadovaný typ.

Po definovaní obsluhy je nutné jej predanie prostredníctvom rozhrania *IModul*, aby mohla byť použitá. Obsluhy pre daný typ udalosti sa predávajú prostredníctvom metódy `VratGenerujKodZaciatok`, a to nasledovne.

```
public List<DelegatGenerujKod> VratGenerujKodZaciatok()
{
      List<DelegatGenerujKod> obsluhaUdalosti =
            new List<DelegatGenerujKod>();

      obsluhaUdalosti.Add(
            new DelegatGenerujKod(typeof(InitialNode),
            GenerujKodZaciatok_InitialNode)
      );

      return obsluhaUdalosti;
}
```

Obsluha sa predáva ako prvok zoznamu `obsluhaUdalosti`, pričom je nutné definovať typ elementu na ktorý sa vzťahuje. Pre zviazanie typu elementu a implementácie obsluhy bola v prípade tejto udalosti zavedená trieda `DelegatGenerujKod`.

Samotné vykonanie obsluhy udalosti prebieha nasledovne. Prostredníctvom rozhrania *IModul* sú z modulu získané všetky obsluhy, ktoré sú následne priradené do eventov elementov, na ktoré sa vzťahujú. Pri prechode vytvorenou grafovou reprezentáciou (abstraktným syntaktickým stromom) zabezpečí vyššie popísaný grafový algoritmus jej vyvolanie, ak bola definovaná. Popis

jednotlivých udalostí a príslušných delegátov je obsiahnutý v rámci dodanej programátorskej dokumentácie.

6.5. Popis pomocných tried

Cieľom tejto sekcie je popis jednotlivých pomocných tried, ktoré je možné využiť pri generovaní kódu. Tie sú obsiahnuté v nasledujúcich súboroch (ich umiestnenie viď. sekcia *"Štruktúra projektu"*)

- *Nazvy.cs* – obsahuje statickú triedu *Nazvy* umožňujúcu generovanie jednoznačných názvov, zisťovanie duplicít, normalizáciu názvov pre použitie v programovacích jazykoch, uchovávanie názvov výstupných tried a objektov jednotlivých elementov.
- *Triedy.cs* – obsahuje triedy umožňujúce uloženie metadát o pomocných triedach uchovávajúcich výstupy elementov.
- *Vystup.cs* – obsahuje statickú triedu *Vystup* umožňujúcu zobrazenie štandardného výstupu alebo chybovej hlášky pri spracovaní vstupného diagramu aktivity, ako i uloženie vygenerovaného kódu do cieľového súboru.

6.6. Definícia nového elementu

Zvolená architektúra umožňuje okrem definovania nového modulu taktiež i začlenenie nového elementu do procesu generovania kódu od jeho extrahovania z XMI až po samotné generovanie. Pridanie nového elementu obnáša nasledujúce kroky

- Definovanie triedy reprezentujúcej daný element. Jej predkom musí byť trieda `Element`, aby bolo možné daný element začleniť do reprezentácie diagramu aktivity prostredníctvom repository. V prípade definovania nového typu akcie je ako predka vhodné použiť triedu `Action`.
- Implementácia metódy `VlastneSpracovanieXMI(XmlTextReader reader)` (rozhranie *IModul*), ktorej vstupom je premenná `reader` obsahujúca aktuálne načítaný XML element zo vstupného súboru. V rámci tejto metódy je nutné vytvoriť inštanciu odpovedajúcej triedy, ktorá bude reprezentovať načítaný element. Vytvorenú inštanciu je následne nutné začleniť do diagramu aktivity. V prípade akcie by začlenenie bolo zrealizované nasledujúcim volaním metódy `ad.PridajAkciu(instancia)`.
- Nakoniec ostáva definovanie udalostí, ktoré sa majú pri generovaní kódu z pridávaného elementu vyvolať (viď. popis rozhrania *IModul*).

7. Alternatívne prístupy pri generovaní kódu

Cieľom tejto kapitoly je popis niektorých z alternatívnych prístupov pri generovaní kódu pomimo vyššie analyzovanej metódy, ktorá bola použitá pri implementácii prototypu. Pozornosť bude zameraná taktiež na ich jednotlivé výhody a nevýhody. Konkrétne budú rozobrané nasledujúce metódy, ktoré patria medzi najznámejšie a najpoužívanejšie

- Priamy prepis do cieľového kódu.
- Generovanie kódu použitím XSL transformácií.
- Generovanie kódu použitím gramatík.

Jednotlivé prístupy môžeme rozdeliť do dvoch hlavných kategórií

- *Relačný prístup* – je založený na definovaní relácií medzi zdrojovou a cieľovou reprezentáciou. Do tejto kategórie môžeme zaradiť prvý zo spomínaných prístupov. Vo svojej podstate sa jedná o definovanie mapovania medzi elementmi vstupnej a výstupnej reprezentácie. Hlavnou výhodou tejto metódy je jej jednoduchosť a priamočiarosť. Nevýhodou je, že tento prístup je možné použiť iba v prípade, ak každý prvok vstupnej reprezentácie je možné jednoznačne reprezentovať odpovedajúcim prvkom výstupnej reprezentácie bez ohľadu na kontext, v ktorom je použitý. To je typicky možné u diagramov popisujúcich statickú štruktúru akými sú napr. diagramy tried. Pre generovanie z diagramov aktivity je tento prístup nemožný.
- *Operačný prístup* – je založený na definovaní procesu transformácie medzi zdrojovou a cieľovou reprezentáciou. Do tejto kategórie môžeme zaradiť zvyšné dva prístupy, teda generovanie kódu použitím XSL transformácií alebo gramatík. V nasledujúcom texte budú bližšie rozobrané obidve z týchto metód.

7.1. Generovanie kódu použitím XSL transformácií

XSLT (eXtensible Stylesheet Language Transformations) je značkovací jazyk konzorcia W3C umožňujúci transformáciu XML dokumentov na základe definovaných pravidiel. Tie sú zapisované prostredníctvom XML notácie. Je často používaný na konverziu XML dát do HTML alebo XHTML dokumentov. Samotná transformácia prebieha nasledovne

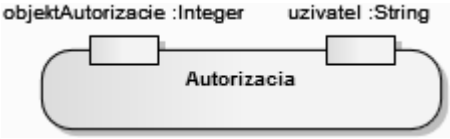
- Vstupom procesu transformácie je jeden alebo viac XML dokumentov a popis samotných XSL transformácií typicky prostredníctvom šablóny.
- Následne XSLT procesor aplikuje dané transformácie na vstupných XML súboroch.

- Výstupom procesu sú nové dokumenty, pričom pôvodné ostanú nezmenené.

XSLT procesor sa často vyskytuje ako samostatná aplikácia, prípadne môže byť súčasťou pokročilejšieho XML editora. Jeho implementácia je taktiež súčasťou mnohých vyšších programovacích jazykov ako napr. Java, C# vo forme odpovedajúcich tried. Typicky prijíma jeden vstupný XML súbor a jeden popis transformácií. Keďže vstupom, na ktorom sú aplikované XSL transformácie, je XML dokument, je možné použiť priamo export príslušného CASE nástroja do formátu XMI. Prípade ten je možné ešte dodatočne zjednodušiť a samotné transformácie generujúce cieľový kód použiť až na túto upravenú reprezentáciu. Výhodou takéhoto prístupu je zjednodušenie šablón generujúcich kód a efektívnejšie riadenie zmien. Ďalšou alternatívou môže byť definovanie metamodelu, do ktorého reprezentácie budú jednotlivé modely prevádzané (z formátu XMI). Generovanie kódu bude následne realizované z popisu v danom metamodely. Vyplývajúcou výhodou je, že pre zavedenie generovania z nového typu UML diagramu je nutné iba definovane jeho prevodu do reprezentácie v danom metamodely.

Príklad

Diagram zobrazujúci aktivitu obsahujúcu dva vstupné parametre.



Reprezentácia daného diagramu prostredníctvom XMI.

```
<?xml version="1.0" encoding="windows-1252"?>

<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">

<xmi:Documentation exporter="Enterprise Architect" exporterVersion="6.5"/>
<uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
<packagedElement xmi:type="uml:Package"
xmi:id="EAPK_70BB35C7_096E_4cd8_9E38_99E39958A2F2" name="Use Case Model"
visibility="public">

<packagedElement xmi:type="uml:Activity"
xmi:id="EAID_79274BE1_E242_4300_9F51_CD1DB1B8E450" name="Autorizacia"
visibility="public">
```

```
<node xmi:type="uml:ActivityParameterNode"
xmi:id="EAID_B66B284B_DA1B_4b84_994F_B53986D5752E" name="objektAutorizacie"
visibility="public" direction="in">
<type xmi:type="uml:PrimitiveType"
href="http://schema.omg.org/spec/UML/2.1/uml.xml#Integer"/>
</node>
<node xmi:type="uml:ActivityParameterNode"
xmi:id="EAID_007801F3_4E99_4038_98CC_E61910D426E8" name="uzivatel"
visibility="public" direction="in">
<type xmi:type="uml:PrimitiveType"
href="http://schema.omg.org/spec/UML/2.1/uml.xml#String"/>
</node>
</packagedElement>

</packagedElement>
</uml:Model>

</xmi:XMI>
```

Definícia XSLT šablóny, ktorá bude použitá pri transformácii. Zabezpečuje vygenerovanie definície metódy vrátane zoznamu vstupných parametrov.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs"
version="2.0" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">

<xsl:template match="/">
<xsl:for-each select="//packagedElement[@xmi:type='uml:Activity']">
<xsl:value-of select="@visibility"/> void <xsl:value-of select="@name"/>
(<xsl:for-each select="//node[@xmi:type='uml:ActivityParameterNode' and
@direction='in']">
<xsl:value-of select="concat(substring(type/@href, 44, string-
length(type/@href) - 43), ' ')"/><xsl:value-of select="@name"/>
<xsl:if test="position() != last()">, </xsl:if>
</xsl:for-each>)
{
      throw new NotImplementedException();
}
</xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Výstup XSLT procesora pre danú šablónu a diagram reprezentovaný prostredníctvom XMI.

```
public void Autorizacia (Integer objektAutorizacie, String uzivatel)
{
      throw new NotImplementedException();
}
```

7.1.1. Výhody

- Nenáročná implementácia.
- Nie je nutný žiadny špeciálny softvér, keďže XSLT procesor je súčasťou bežne používaných jazykov ako Java, C#. Nadefinované šablóny je teda možné aplikovať prostredníctvom na to určených tried.

7.1.2. Nevýhody

- XSLT nie je založené na matematickej formalizácii na rozdiel od grafov alebo gramatík.
- Spracovanie XSL transformácií je neefektívne v prípade, ak vstupná reprezentácie nie je vo forme stromu ale obecného grafu. K takejto situácii dochádza v diagramoch aktivít v prípade modelovania cyklov.
- Zložité XSL transformácie sú neprehľadné, ťažko udržiavateľné a modifikovateľné.

7.2. Generovanie kódu použitím gramatík

Ide o prístup opierajúci sa o teóriu z oblasti automatov a gramatík. Ten je založený na definovaní zobrazenia medzi jazykmi popisujúcimi diagram aktivity a konkrétny cieľový jazyk. Majme vstupný jazyk L_{in} generovaný gramatikou G_{in} a popisujúci diagram aktivity vo formáte XMI. Nech jazyk L_{out} generovaný gramatikou G_{out} alebo prijímaný automatom A_{out} je výstupný a popisujúci cieľový jazyk. Potom samotný preklad bude realizovaný prostredníctvom zobrazenia $L_{in} \rightarrow L_{out}$, kde $\forall\, w_{in} \in L_{in}\ \exists\, w_{out} \in L_{out}$ a pre $w_{in} \notin L_{in}$ zobrazenie neexistuje. Keďže ako UML, tak i programovacie jazyky sú bezkontextové jazyky, je možné ich generovať bezkontextovými gramatikami. Nasleduje popis týchto gramatík.

Bezkontextovou gramatikou G nazývame štvoricu $G = (V_N, V_T, S, P)$, kde

- V_N je konečná množina neterminálnych symbolov.
- V_T je konečná množina terminálnych symbolov.
- S je počiatočný neterminálny symbol, $S \in V_N$.
- P je prepisovacím systémom obsahujúcim konečný počet prepisovacích pravidiel tvaru $X \rightarrow w$, kde $X \in V_N$, $w \in (V_N \cup V_T)^*$.

Aplikáciou jednotlivých prepisovacích pravidiel na počiatočný neterminálny symbol S je možné získať všetky slová, ktoré daná gramatika generuje. Jazyk L(G) generovaný gramatikou G definujeme nasledovne L(G) = {w | w ∈ V_T* & S ⇒* w}, kde zápis ⇒* označuje prepis neterminálneho symbolu S na slovo w aplikáciou prepisovacích pravidiel.

Príklad

Diagram zobrazujúci aktivitu obsahujúcu dva vstupné parametre.

Nasleduje definovanie gramatík popisujúcich vstupný a výstupný jazyk a zobrazenia medzi nimi. Gramatiky nie sú kompletné a slúžia len ako ilustrácia možného riešenia. Ich cieľom je popis časti jazyka slúžiacej na zápis aktivity vrátane jej vstupných parametrov.

Návrh gramatiky G_{in} = (V_N, V_T, S, P) popisujúcej vstupný jazyk

- V_N = {S_{in}, $Aktivity_{in}$, $Aktivita_{in}$, $Viditelnost_{in}$, $NavratovyTyp_{in}$, $Parametre_{in}$, $Parameter_{in}$, Typ_{in}}.
- V_T = { <, >, packagedElement, xmi, :, type, ", uml, Activity, id, =, name, visibility, /, public, private, protected, node, ActivityParameterNode , direction, in, PrimitiveType, href, http://schema.omg.org/spec/UML/2.1/uml.xml , #}.
- S_{in} – počiatočný neterminálny symbol.
- P - systém prepisovacích pravidiel majúci nasledujúci tvar

S_{in} → $Aktivity_{in}$
$Aktivity_{in}$ → $Aktivita_{in}$ $Aktivity_{in}$
$Aktivity_{in}$ → $Aktivita_{in}$
$Aktivita_{in}$ → **<packagedElement xmi:type="uml:Activity" xmi:id="** *Id_{in}* **" name="** *$Nazov_{in}$* **" visibility="** $Viditelnost_{in}$ **"></packagedElement>**
$Aktivita_{in}$ → **<packagedElement xmi:type="uml:Activity" xmi:id="** *Id_{in}* **" name="** *$Nazov_{in}$* **" visibility="** $Viditelnost_{in}$ **">** $Parametre_{in}$ **</packagedElement>**
$Viditelnost_{in}$ → **public**
$Viditelnost_{in}$ → **private**
$Viditelnost_{in}$ → **protected**
$Parametre_{in}$ → Parameter Parametre

```
Parametre_in → Parameter
Parameter_in → <node xmi:type="uml:ActivityParameterNode" xmi:id=" Id_in "
               name=" Nazov_in " visibility=" Visibility_in "
               direction="in"><type xmi:type="uml:PrimitiveType"
               href="http://schema.omg.org/spec/UML/2.1/uml.xml#
               Typ_in "/></node>
Typ_in → String
Typ_in → Integer
```

Poznámka : $Nazov_{in}$, Id_{in} reprezentujú textové literály obsahujúce názov a identifikátor aktivity alebo vstupného parametru.

Návrh gramatiky $G_{out} = (V_N, V_T, S, P)$ popisujúcej výstupný jazyk

- V_N = {S_{out}, $Aktivity_{out}$, $Aktivita_{out}$, $Viditelnost_{out}$, $NavratovyTyp_{out}$, $Parametre_{out}$, $Parameter_{out}$, Typ_{out}}.
- V_T = {public, private, protected, void, String, Integer, {, }, (,), throw, new, NotImplementedException }.
- S_{out} – počiatočný neterminálny symbol.
- P - systém prepisovacích pravidiel majúci nasledujúci tvar

```
S_out → Aktivity_out
Aktivity_out → Aktivita_out Aktivity_out
Aktivity_out → Aktivita_out
Aktivita_out → Viditelnost_out NavratovyTyp_out Nazov_out (Parametre) { throw new
               NotImplementedException(); }
Viditelnost_out → public
Viditelnost_out → private
Viditelnost_out → protected
NavratovyTyp_out → void
Parametre_out → Parameter_out , Parametre_out
Parametre_out → Parameter_out
Parameter_out → Typ_out Nazov_out
Typ_out → String
Typ_out → Integer
```

Poznámka : $Nazov_{out}$ reprezentuje textový literál obsahujúci názov aktivity alebo vstupného parametru.

Zobrazenie medzi jazykmi generovanými gramatikami G_{in} a G_{out} bude definované prostredníctvom gramatiky H, ktorá z G_{in} preberie neterminálne symboly a z G_{out} terminálne symboly. Inými slovami z počiatočného neterminálneho symbolu S_{in} gramatiky G_{in} sa budú aplikáciou prepisovacích pravidiel generovať slová odpovedajúce gramatike G_{out}.

Návrh gramatiky H = (V_N, V_T, S, P)

- $V_N = V_N$ gramatiky G_{in}.
- $V_T = V_T$ gramatiky G_{out}.
- S – počiatočný neterminálny symbol gramatiky G_{in}.
- P - systém prepisovacích pravidiel majúci nasledujúci tvar

```
Sin → Aktivityin
Aktivityin → Aktivitain Aktivityin
Aktivityin → Aktivitain
Aktivitain → Viditelnostin NavratovyTypin Nazovin (Parametre) { throw new
             NotImplementedException(); }
Viditelnostin → public
Viditelnostin → private
Viditelnostin → protected
NavratovyTypin → void
Parametrein → Parameterin , Parametreout
Parametrein → Parameterin
Parameterin → Typin Nazovin
Typin → String
Typin → Integer
```

Poznámka : *$Nazov_{in}$* reprezentuje textový literál obsahujúci názov aktivity alebo vstupného parametru.

7.2.1. Výhody

- Prístup opierajúci sa o teóriu z oblasti automatov a gramatík. Formalizácia gramatík umožňuje použitie voľne dostupných generátorov parserov, čím odpadá množstvo náročnej implementácie.

7.2.2. Nevýhody

- Náročnejšie riešenie vyžadujúce vedomosti v danej oblasti.

8. Prototyp nástroja

Cieľom tejto kapitoly je popis užívateľského rozhrania a práce s prototypom nástroja, ktorý je výsledkom diplomovej práce. Ten je súčasťou dodaného DVD, ktorého obsah je nasledovný

- Adresár *„Text“* obsahuje text diplomovej práce v elektronickej podobe (formát docx a pdf).
- Adresár *„Web“* obsahuje webovú stránku, ktorá slúžila na zverejňovanie priebežných výsledkov práce.
- Adresár *„Prototyp“* obsahuje generátor kódu, ktorý je výsledkom tejto diplomovej práce.
- Adresár *„Generovane situacie“* obsahuje vytvorené projekty v rámci CASE nástroja Enterprise Architect. Tie obsahujú modelované situácie používane pri vývoji a testovaní prototypu. Modelované situácie zároveň slúžia ako referencia o možnostiach generátora.
- Adresár *„Zdrojovy kod“* obsahuje samotný projekt Visual Studia, v ktorom bol prototyp vyvinutý.
- Adresár “*Programatorska dokumentacia”* obsahuje dokumentáciu zdrojového kódu vo formáte html popisujúcu jednotlivé triedy a ich metódy. Dokumentácia bola vygenerovaná prostredníctvom nástroja Doxygen.

Nasleduje popis inštalácie a užívateľského rozhrania prototypu.

8.1. Inštalácia

Samotná inštalácia prebieha nakopírovaním adresára obsahujúceho prototyp na ľubovoľné miesto na disku. Aplikácia beží pod operačným systémom Windows XP, Vista alebo 7, pričom je nutné mať nainštalovaný .NET Framework verzie 3.5 alebo vyššej.

8.2. Užívateľské rozhranie

Po spustení prototypu sa zobrazí obrazovka určená pre samotné generovanie kódu. V jej ľavej časti je umiestnené menu slúžiace na prechod medzi jednotlivými časťami aplikácie, ktorými sú

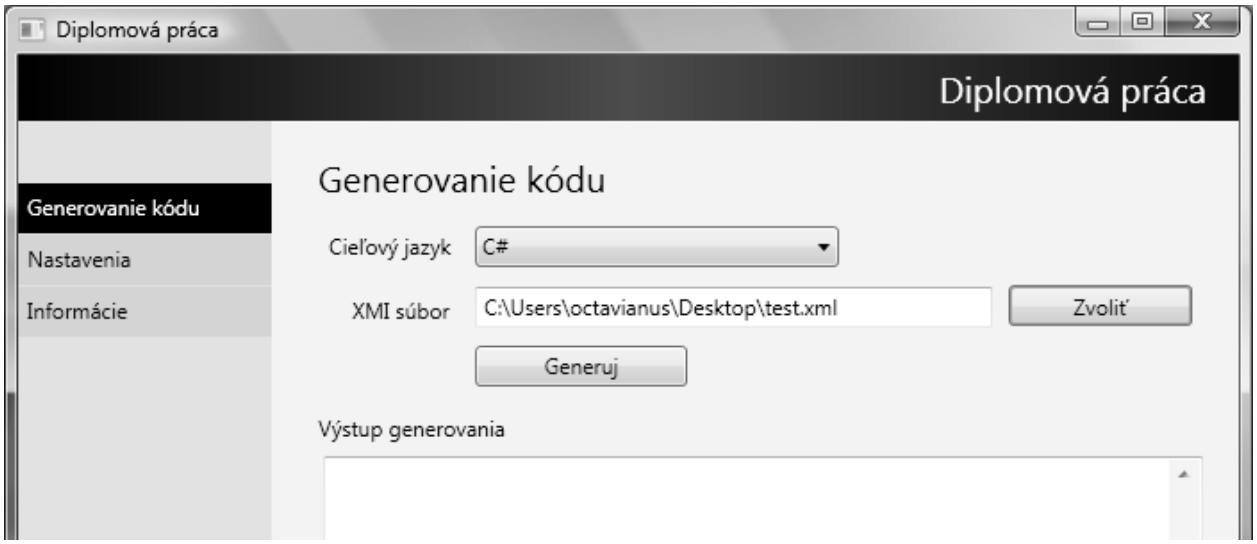
- Generovanie kódu
- Nastavenia
- Informácie

Nasleduje popis jednotlivých častí prototypu nástroja.

8.2.1. Generovanie kódu

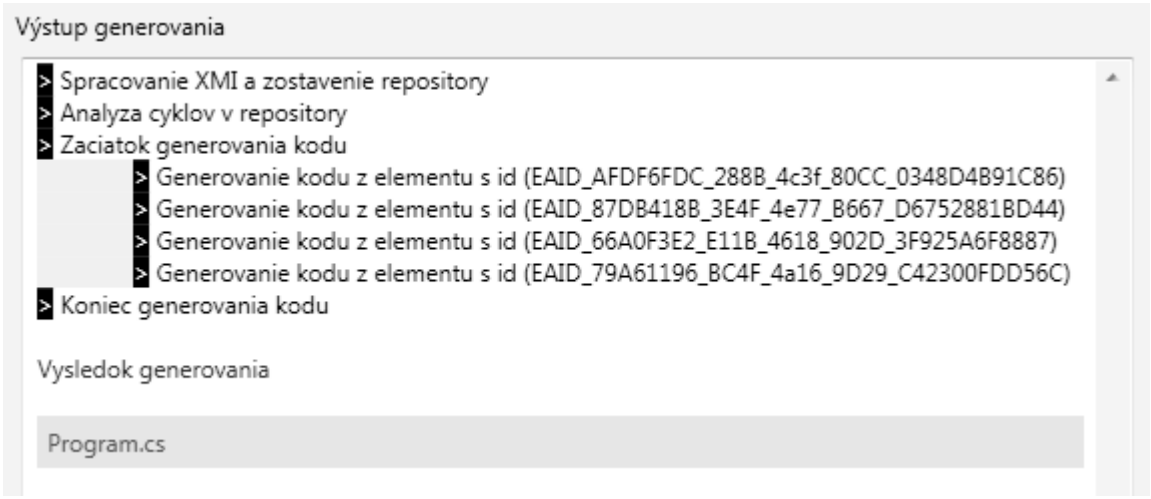
Táto časť aplikácie obsahuje rozhranie pre realizáciu samotného generovania kódu. Umožňuje zadanie nasledujúcich údajov

- *Cieľový jazyk* – obsahuje zoznam všetkých implementovaných modulov v rámci aplikácie. Jeho zvolením sa určí, ktorý z modulov má byť pri generovaní použitý. Prototyp aktuálne obsahuje jedine modul umožňujúci generovanie do jazyka C#. Modul Java bol pridaný iba za účelom prezentovania modularity prototypu.
- *XMI súbor* – obsahuje súbor nesúci export diagramu aktivity vo formáte XMI verzie 2.1 (popis získania exportu z nástroja Enterprise architect viz. *"Príloha A"*). Zadáva sa prostredníctvom dialógového okna vyvolaného po stlačení tlačítka *Zvoliť,* prípadne priamym vpísaním cesty k súboru.

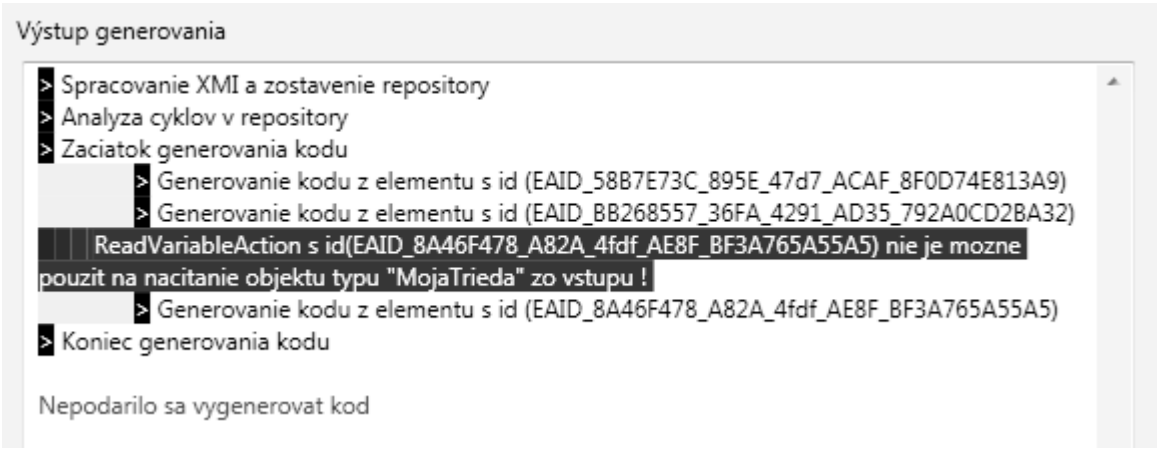


Samotné generovanie kódu sa zrealizuje stlačením tlačítka *Generuj*. V prípade, ak niektorý so vstupných parametrov nie je korektne zadaný (nezadaný alebo neexistujúci XMI súbor, cieľový adresár) program zobrazí príslušnú chybovú hlášku. Jednotlivé kroky generovania sú zobrazené vo výstupe generovania. V prípade, že generovanie prebehlo bez problémov je na konci výstupu

zobrazený zoznam odkazov na súbory obsahujúce cieľový kód. Po kliknutí na niektorý z nich bude príslušný súbor otvorený operačným systémom.

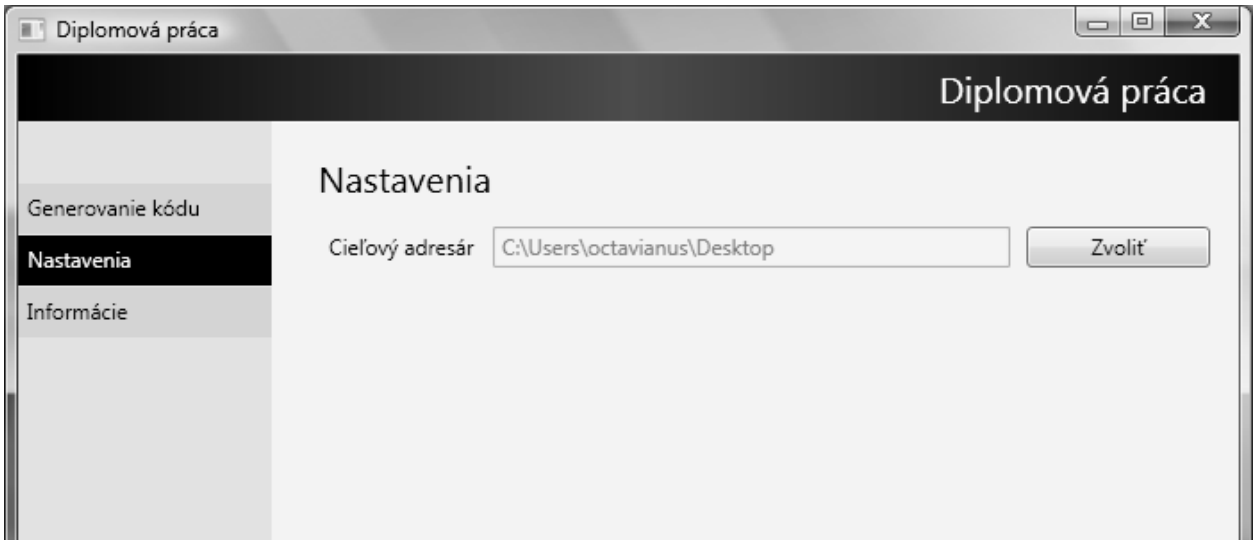


V prípade výskytu chyby pri generovaní je na výstupe zobrazená príslušná chybová hláška. Tá je farebne odlíšená pre lepšie rozoznanie. Zároveň nie je vygenerovaný žiaden súbor.



8.2.2. Nastavenia

Táto časť aplikácie obsahuje nastavenie cieľového adresára, do ktorého budú umiestnené všetky vygenerované súbory. Po kliknutí na tlačítko *Zvoliť* dôjde k vyvolaniu dialógového okna, v ktorom je možné zvoliť nový cieľový adresár. Po potvrdení voľby sa nové nastavenia uložia v rámci aplikácie a daný adresár bude použitý i pri opätovnom spustený. Pri samotnom generovaní dôjde k nahradeniu všetkých súborov v cieľovom adresári súbormi vygenerovanými prototypom v prípade, ak majú rovnaký názov.



8.2.3. Informácie

Táto časť aplikácie obsahuje stručný prehľad informácii o diplomovej práci.

9. Existujúce nástroje

Cieľom tejto kapitoly je porovnanie CASE nástrojov z hľadiska podpory pri generovaní kódu z diagramov aktivity. Zameriame sa iba na tie z nich, ktoré umožňujú generovanie z aspoň niektorého druhu UML diagramov. Nasledujúce tabuľka poskytuje prehľad najpoužívanejších CASE nástrojov a ich schopnosti generovať kód z diagramov tried a z jednotlivých druhov diagramov správania. Symbol * v príslušnom stĺpci znamená, že daný nástroj poskytuje podporu pri generovaní kódu z príslušného druhu diagramu. Zahrnutie diagramov tried do prehľadu je z dôvodu, že ich porozumenie je predpokladom pre generovanie z diagramov správania v rozumnom rozsahu, keďže tie môžu pracovať s ich inštanciami.

CASE nástroj	*Diagramy tried*	*Sekvenčné diagramy*	*Stavové diagramy*	*Diagramy aktivity*
Enterprise Architect	*	*	*	*
Visual Paradigm	*	*	*	
Altova UModel	*	*	*	
ArgoUML	*	*	*	
Poseidon for UML	*		*	
UML lab	*	*		
MagicDraw	*	*		
Sybase PowerDesigner	*			
Umbrello UML modeler	*			
MS Visio	*			
Objecteering UML Modeler	*			
StarUML	*			
IBM - Rational Software Modeler	*			
Select component architect	*			
AgileJ Structure Views	*			

Z uvedenej tabuľky vyplýva, že najčastejším druhom UML diagramov, z ktorého je generovaný kód, je diagram tried. Je to spôsobené predovšetkým nenáročnou implementáciou, ktorú je možné realizovať jednoduchým prepisom. Pokročilejšie CASE nástroje navyše umožňujú generovanie zo sekvenčných a stavových diagramov. Nástoj *Enterprise architect* ako jediný

umožňuje generovanie z diagramov aktivity. V nasledujúcom texte sa zameriame práve naň a popíšeme jeho hlavné vlastnosti v analyzovanej oblasti.

9.1. Enterprise Architect

Je pokročilý CASE nástroj umožňujúci ako modelovanie UML diagramov, tak i generovanie kódu do zvoleného cieľového jazyka. Generovanie kódu je v EA realizované podobne ako v iných nástrojoch, a to prostredníctvom tzv. šablón. Ich obsahom je postupnosť príkazov v jazyku syntaxou pripomínajúcom zápis XSLT transformácií. Jazyk obsahuje iba niektoré základné konštrukcie, akými sú napríklad definovanie premenných jednoduchých typov, vykonávanie aritmetických operácií, zápis podmienok a podobne. Vstupom šablóny je zoznam atribútov vzťahujúcich sa na daný element, ktoré je možné použiť. Výstupom je text reprezentujúci cieľový kód pre daný element. Nasleduje zhrnutie jednotlivých charakteristík generovania kódu z diagramov aktivity v tomto nástroji.

- Šablóny sa definujú pre každý cieľový jazyk zvlášť, pričom jednotlivé cieľové jazyky je možné pridávať.
- V šablónach nie je možné definovať cyklus, a teda spracovanie zoznamu elementov je možné len prostredníctvom rekurzie (zavolanie tej iste šablóny).
- Zápis prostredníctvom definovaného jazyka je neprehľadný a ťažko čitateľný.
- Generovanie kódu je možné iba z akcií, ale už nie zo samotných elementov UML. Elementy ako napr. decision/merge node (ktoré nemajú odpovedajúcu reprezentáciu prostredníctvom akcií) sú reprezentované ako akcie “If”, “Loop” (nedefinované v špecifikácii UML).
- Generátor obsahuje definície šablón len pre niektoré druhy akcií, avšak je možné ich doplnenie o ďalšie.
- Generátor nerealizuje syntaktickú alebo sémantickú analýzu, teda výrazy zadané napr. v rámci *guard expressions* nie sú kontrolované, ale iba kopírované priamo na výstup. Tým môže dôjsť k vygenerovaniu nekorektného kódu.

Záverom je možné zhrnúť, že generátor kódu EA je modulárny, teda je možné ho rozšíriť o ďalšie cieľové jazyky a zároveň je možné upraviť definície existujúcich šablón podľa potreby. Najväčšou nevýhodou generátoru je fakt, že dokáže generovať kód iba z diagramu aktivity definovaného prostredníctvom repository (akcií), prípadne je možné použitie tých elementov, ktoré nemajú priamy prepis do akcií. Vhodnejším prístupom je umožniť užívateľovi modelovanie diagramu použitím ľubovoľných elementov, ten následne reprezentovať prostredníctvom repository a až túto reprezentáciu použiť pri samotnom generovaní. Vytvorenie modelu prostredníctvom repositiry je totiž náročné a v prípade rozsiahlych modelov neprehľadné (viď. kapitola *„Prevod diagramu aktivity do repository”*), čím samotné generovanie kódu stráca na význame.

10. Záver

Cieľom tejto kapitoly je zhodnotenie vytýčených cieľov v úvode práce a popis možných vylepšení prototypu do budúcnosti.

10.1. Stanovené ciele

V tejto sekcii zhrnieme závery pre vytýčené ciele z úvodu, ktorými sú

- Analýza možností generovania kódu z diagramov aktivity.
- Implementácia prototypu nástroja umožňujúceho generovanie kódu z diagramov aktivity.

10.1.1. Analýza možností generovania kódu

Kód z diagramov aktivity je možné generovať viacerými prístupmi. Ich podrobný popis je uvedený v kapitolách *"Vlastné generovanie kódu"* a *„Alternatívne prístupy pri generovaní kódu"*. Zhrnutím môžeme povedať, že základnými metódami je generovanie prostredníctvom XSLT transformácii, aplikáciou gramatík (klasický prístup z teórie prekladačov) alebo generovaním pri prechode vhodnou grafovou reprezentáciou diagramu. Posledný prístup bol uplatnený pri implementácii prototypu. Samotné generovanie bolo možné už z popisu UML verzie 1.0, ale až UML verzie 2.0 obohatilo diagramy aktivity o prvky, vďaka ktorým je generovanie vhodné a užitočné. Medzi ne v prvom rade patria akcie, prostredníctvom ktorých je definovaná reprezentácia (slúžiaca ako medzikód) medzi konkrétnym diagramom a jeho reprezentáciou v cieľovom programovacom jazyku. Ďalším významným obohatením je element ActivityParameterNode reprezentujúci konkrétny vstup alebo výstup aktivity. Spolu s akciami slúžiacimi na vytváranie objektov konkrétneho typu a manipuláciu s ich hodnotami umožňujú generovanie kompletného programu a nie len jeho kostry. Nevýhodou pri vytváraní detailného správania je náročnosť samotného modelovania, pričom rozsiahly model môže v konkrétnom programovacom jazyku reprezentovať iba nepatrnú časť kódu.

10.1.2. Implementácia prototypu nástroja

Súčasťou DVD dodaného s diplomovou prácou je prototyp nástroja umožňujúceho generovanie kódu z elementov diagramu aktivity s nasledujúcimi vlastnosťami

- Vstupom je diagram aktivity exportovaný do XMI formátu verzie 2.1..
- Generovanie kódu je realizované prevodom vstupného diagramu do reprezentácie definovanej MOF metamodelom (viď. kapitola *„Generovanie kódu"*) a jej následným prechodom príslušným grafovým algoritmom.
- Prototyp umožňuje pridávanie modulov realizujúcich generovanie do rôznych cieľových jazykov. Aktuálne obsahuje implementáciu modulu pre výstup v programovacom jazyku C#.
- Udalosťami riadená architektúra prototypu umožňuje implementovať generovania kódu iba z požadovaných typov elementov.
- Prototyp umožňuje definovanie nových UML elementov diagramu aktivity. Konkrétne ich extrahovanie z XMI formátu, začlenenie do grafovej reprezentácie a definovanie udalostí vyvolaných pri generovaní kódu. Tento prístup je vhodný predovšetkým pri zavedení generovania kódu z novej akcie definovanej UML špecifikáciou.

10.2. Možné vylepšenia

V tejto sekcii popíšeme možné vylepšenia stávajúceho stavu prototypu, ktoré by skvalitnili samotný proces generovania z hľadiska užívateľského rozhrania alebo z hľadiska kvality vygenerovaného kódu. Takými to vylepšeniami napríklad sú

- Optimalizácia vygenerovaného kódu. Príkladom môže byť neuchovávanie návratovej hodnoty aktivity v lokálnej premennej, ak je použitá iba na jednom mieste. Volanie metódy reprezentujúcej danú aktivitu je možné použiť priamo na mieste, kde sa očakáva jej výstup.
- Sprostredkovanie koncovému užívateľovi možnosť ovplyvniť tvar vygenerované kódu tým, že mu bude umožnené zadávanie parametrov pre jednotlivé moduly. Jednotlivé parametre definuje programátor konkrétneho modulu, pričom užívateľ ich bude môcť zadávať cez odpovedajúce rozhranie (GUI, konfiguračný XML súbor, ...). Príkladom by mohlo byť rozhodnutie, či v prípade ak aktivita vracia viaceré výstupy bude vygenerovaná inštancia pomocnej návratovej triedy (aktuálny stav), alebo jednotlivé výstupné parametre budú predávané prostredníctvom referencie (operátor *ref* v prípade jazyka C#).
- Zavedenie syntaktickej a sémantickej analýzy do výrazov, ktoré sú aktuálne priamo kopírované na výstup generátora, a teda môže dôjsť k vygenerovaniu nekorektného kódu. Príkladom môžu byť napríklad výrazy zapísané v rámci *guard expresions* výstupných hrán z UML elementu *decision node*. Realizovanou kontrolou môže byť overenie syntaxie zadaného výrazu (keďže špecifikácia UML ju nedefinuje, je nutné zvoliť jej vhodný tvar, prípadne pre každý modul definovať vlastnú syntax). Ďalšou z kontrol môže byť overenie existencie všetkých premenných použitých v danom výraze a spôsob ich použitia na základe ich typu.

11. Zoznam použitej literatúry

[1] Unified Modeling Language: *Superstructure, version 2.0*, Specification By OMG, 2007

[2] Unified Modeling Language: *Infrastructure, version 2.2*, Specification By OMG, 2007

[3] *MOF 2.0/XMI Mapping, version 2.1.1*, Specification By OMG, 2007

[4] Grady Booch, James Rumbaugh, Ivar Jacobson : *The Unified Modeling Language User Guide*. Second edition. Addison Wesley Professional, 2005. ISBN 0-321-26797-4

[5] Jim Arlow, Ila Neustadt : *UML and the Unified Process – Practical object-oriented analysis and design*. Pearson Education, 2002. ISBN 9780201770605

[6] Dan Pilone, Neil Pitman : *UML 2. 0 In A Nutshell*. Second Edition. O'Reilly, 2005. ISBN 978-0596007959

[7] Tom Pender : *UML Bible*. First edition. John Wiley & Sons, 2005. ISBN 0764526049

[8] Jasmine Farhad : *The UML Extension Mechanisms*. University College London, 2002

[9] Doc. Ing. Karel Richta, CSc : *Materiály k prednáške „Softwarové inženýrství“*, 2011

[10] Conrad Bock : *UML without Pictures*, IEEE Computer Society, 2003

[11] Conrad Bock : *Journal of object technology*, ETH Zurich, Chair of Software Engineering, 2005

[12] Conrad Bock : *UML 2 Activity and Action Models* http://www.jot.fm/issues/issue_2003_07/column3/ (2011-07-22)

[13] Dr. Perdita Stevens : *Tool adaptation, extension and integration using XMI*. The UML 2001 Conference, 2001

[14] Oldřich Nouza : *Principy generování zdrojového kódu z UML*. ČVUT v Praze, FJFI, katedra matematiky, 2004

[15] Dániel Varrá - András Pataricza : *UML Action Semantics for Model Tranformation Systems*. Periodica Politechnica Vol. 47, No. 3, pp. 167–186, 2003

[16] *Introduction to Enterprise Architect, UML Modeling Tool* http://www.sparxsystems.com.au/uml_tool_guide/ (2011-07-22)

[17] Jesse Liberty : *Programming C#*. Second edition. O'Reilly, 2002. ISBN 978-0596003098

[18] MSDM training - *Introduction to C# Programming for the Microsoft .NET Platform*. Course Number: 2124A. Microsoft Corporation, 2001

[19] Christian Nagel - Bill Evjen - Jay Glynn - Morgan Skinner - Karli Watson: *Professional C# 2008*. First edition. John Wiley & Sons, 2008. ISBN 978-0470191378

[20] Jiří Kosek : *XML pro každého*. První vydání. Grada Publishing, 2000. ISBN 80-7169-860-1

[21] Irena Mlýnková, Martin Nečaský, Jaroslav Pokorný, Karel Richta : *Materiály k prednáške „Technologie XML“*, 2011

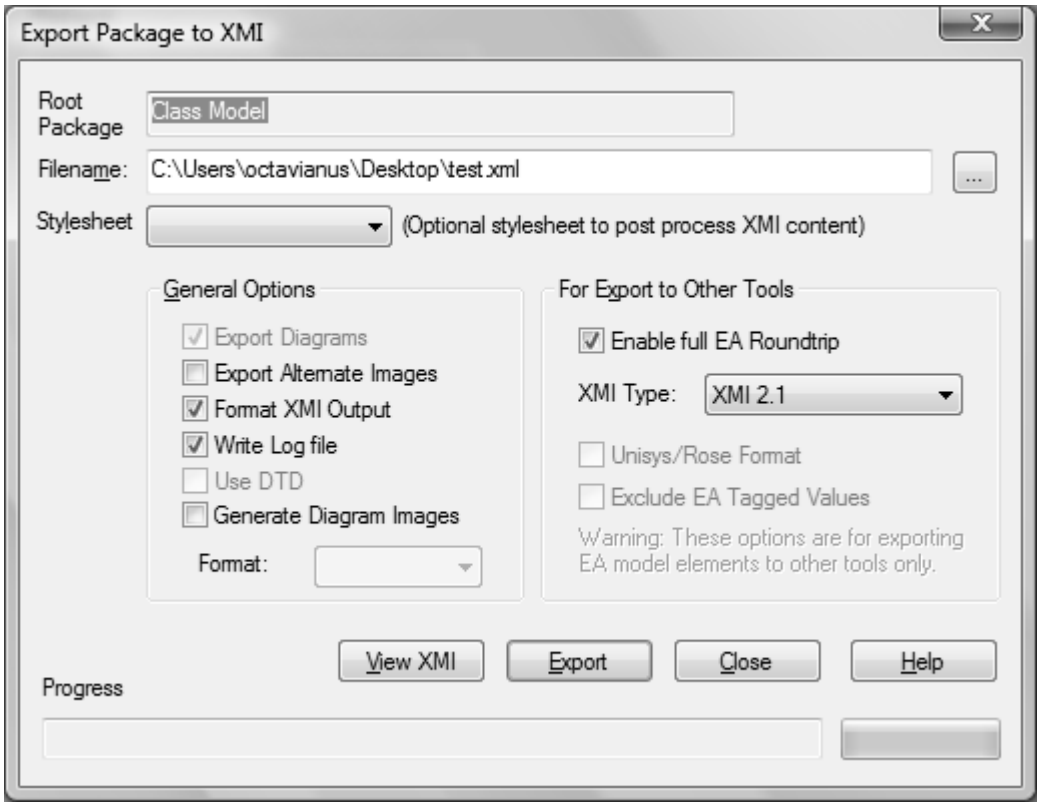
[22] doc. RNDr. Roman Barták, Ph.D. : *Materiály k prednáške „Automaty a gramatiky“*, 2011

[23] RNDr. Jakub Yaghob, Ph.D. : *Materiály k prednáške „Principy překladačů“*, 2011

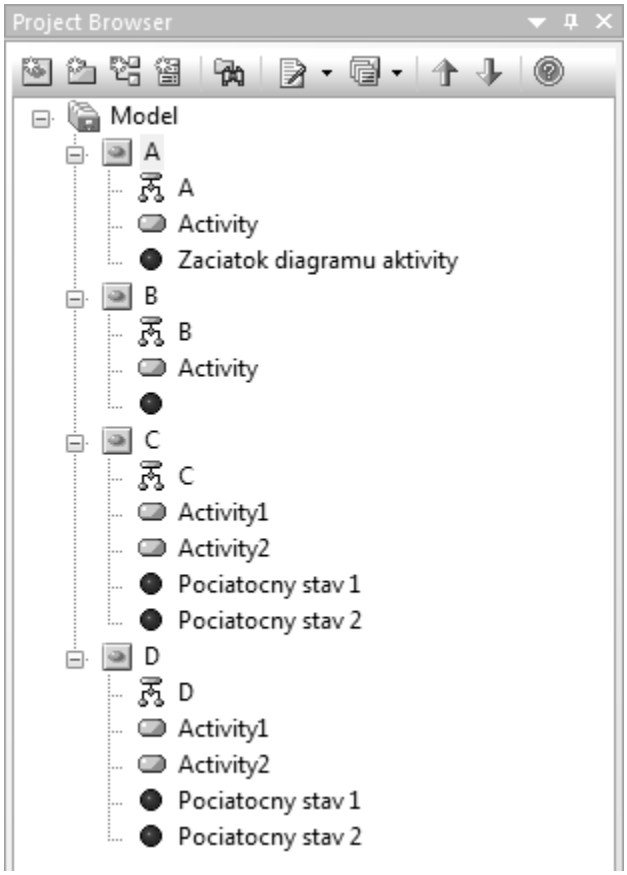
12. Prílohy

12.1. Príloha A

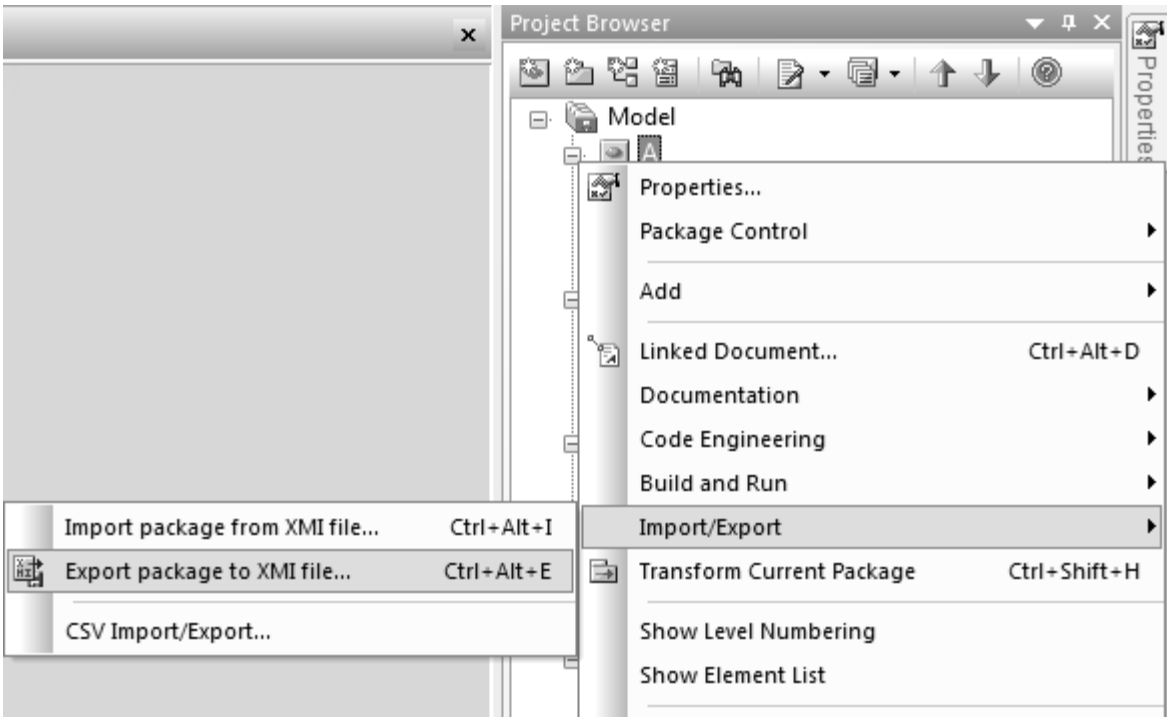
Cieľom tejto prílohy je popis realizácie exportu do formátu XMI v rámci CASE nástroja *Enterprise architect*. Výstupy tohto programu boli používané pri implementácii samotného prototypu. Export je možné po namodelovaní príslušného diagramu aktivity vyvolať prostredníctvom hlavného menu *Project* -> *Import/Export* -> *Export Package to XMI…* alebo prostredníctvom klávesovej skratky *Ctrl+Alt+E*. Pred samotným exportom je zobrazení dialóg umožňujúci definovanie jeho parametrov. Najdôležitejším z nich (pomimo názvu cieľového súboru a jeho umiestnenia) je XMI type, ktorý obsahuje číslo verzie XMI formátu. Pre účely tejto diplomovej práce to musí byť verzia 2.1. Exporty v iných verziách prototyp nedokáže korektne spracovať vzhľadom na ich značné odlišnosti viď. príslušná kapitola o XMI.



Pre účely testovania je možné využiť predpripravené namodelované situácie, ktoré vznikali postupne pri samotnom vývoji prototypu. Tie sú umiestnené na dodanom inštalačnom DVD v adresári *„Generovane situacie“*. Jednotlivé súvisiace modelované situácie sú umiestnené v samostatných súboroch, pričom každá z nich je reprezentovaná samostatným modelom.

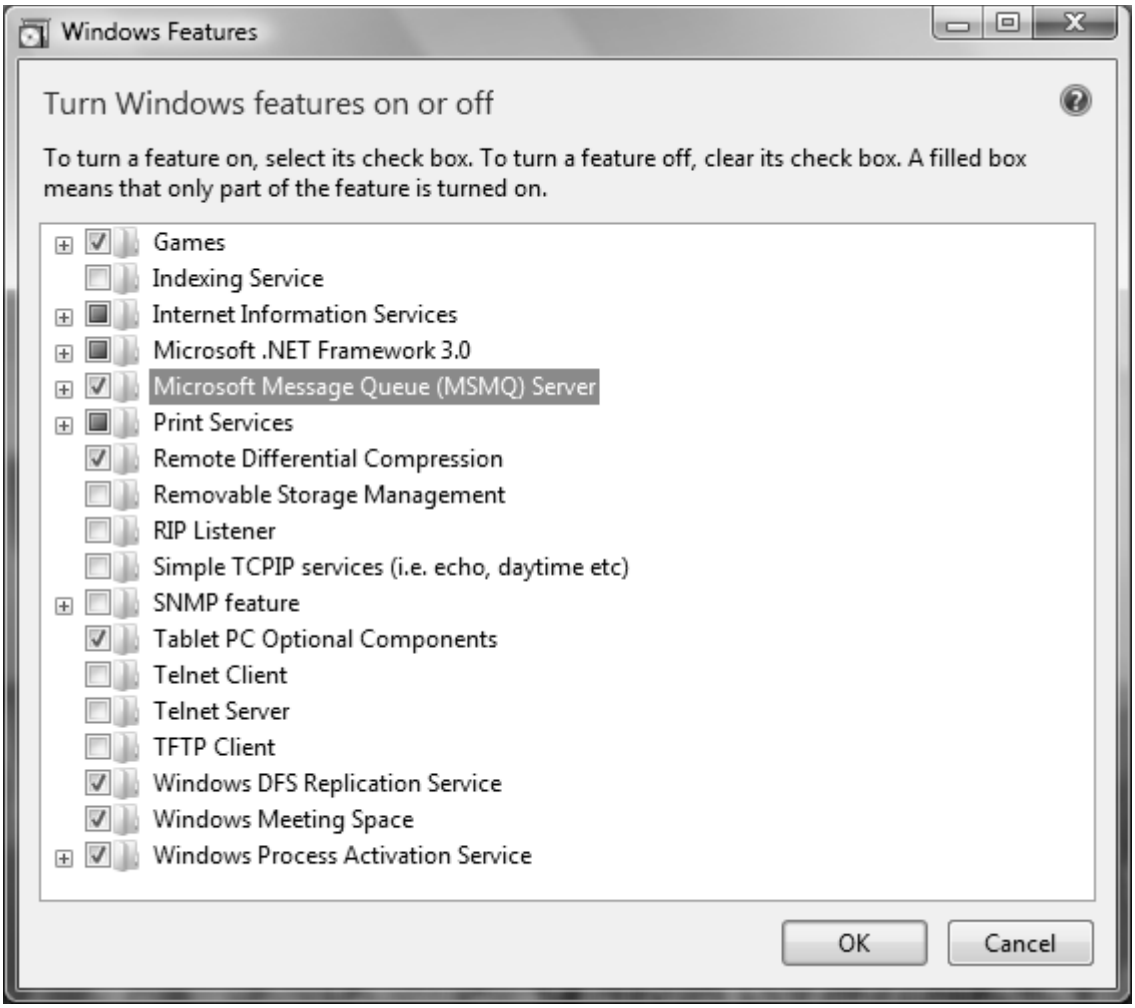


Realizácia exportu len z príslušného modelu je možná napr. prostredníctvom kontextového menu vyvolaného nad daným modelom *Import/Export* -> *Export Package to XMI…*.

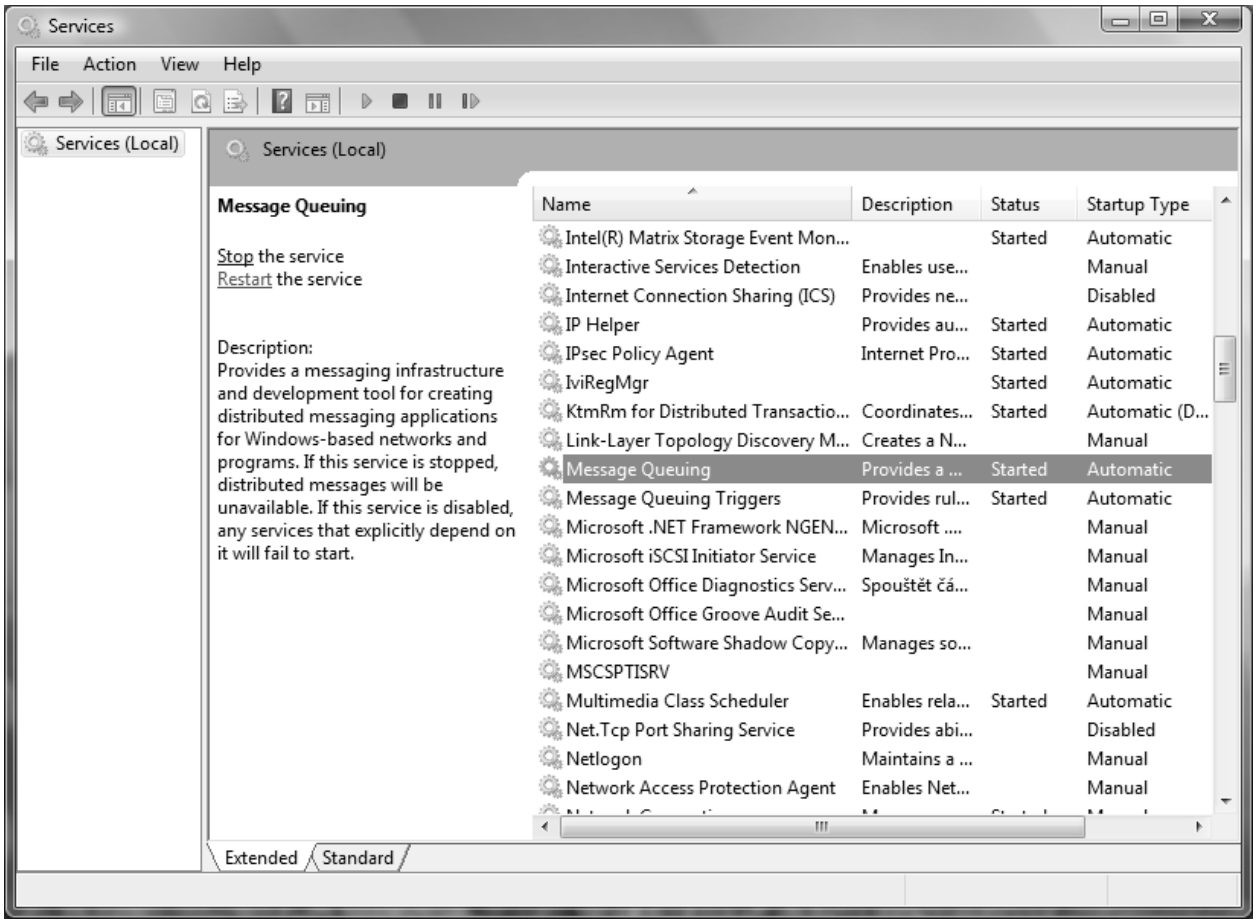


12.2. Príloha B

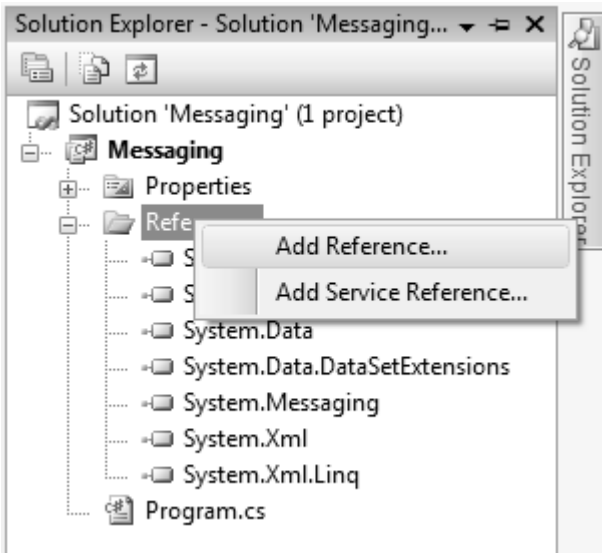
Cieľom tejto prílohy je popis konfigurácie, ktorú je nutné uskutočniť za účelom vygenerovania spustiteľného kódu z UML elementov realizujúcich odoslanie a prijatie signálu. Prototyp využíva pri generovaní z týchto elementov v rámci implementovaného modulu C# službu *Message Queuing* operačného systému Windows. Tá je dostupná v jeho nasledujúcich verziách : Windows Vista, Windows 7 a Windows Server 2008. Keďže tá býva defaultne vypnutá, nasleduje popis jej aktivácie v prostredí operačného systému Windows Vista. V prostrední ostatných operačných systémov je proces rovnaký až na malé rozdiely. Službu *Message Queuing* je nutné povoliť v rámci dialógu dostupného z *Control Panel* `->` *Programs and Features* `->` *Turn Windows features on or off* zaškrtnutím voľby *Microsoft Message Queue (MSMQ) Server.*



To, či sa danú službu podarilo spustiť je možné overiť v dialógu *Services* dostupného napr. z dialógu *Windows Task Manager* (pod záložkou *Services*). Názov služby je *Message Queuing* a jej vykonávanie je vyjadrené hodnotou *Started* v stĺpci *Status*.



Nasleduje popis konfigurácie nástroja *Visual Studio 2008* pre úspešné skompilovanie a spustenie kódu. Po začlenení vygenerovanej triedy prototypom do existujúceho alebo nového projektu je nutné pridanie referencie na *System.Messaging*, aby bolo možné frontu správ použiť. Tú je možné pridať napr. prostredníctvom voľby *„Add Reference ..”* dostupnej z kontextového menu vyvolaného nad zložkou *References* v rámci *Solution Explorer*. Okno Solution Explorer, obsahujúce adresárovú štruktúru a jednotlivé súbory projektu, je dostupné z hlavného menu *View* -> *Solution Explorer* alebo prostredníctvom klávesovej skratky *Ctrl+W, S*.


Solution Explorer - Solution 'Messaging...
Solution 'Messaging' (1 project)
Messaging
Properties
Refe
Add Reference...
Add Service Reference...
System.Data
System.Data.DataSetExtensions
System.Messaging
System.Xml
System.Xml.Linq
Program.cs
Solution Explorer